WIND RIVER

Wind River® Workbench

USER'S GUIDE

3.0

Linux Version
Edition 2

Copyright © 2008 Wind River Systems, Inc.

All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of Wind River Systems, Inc.

Wind River, the Wind River logo, Tornado, and VxWorks are registered trademarks of Wind River Systems, Inc. Any third-party trademarks referenced are the property of their respective owners. For further information regarding Wind River trademarks, please see:

http://www.windriver.com/company/terms/trademark.html

This product may include software licensed to Wind River by third parties. Relevant notices (if any) are provided in your product installation at the following location: *installDir*/*product_name*/**3rd_party_licensor_notice.pdf**.

Wind River may refer to third-party documentation by listing publications or providing links to third-party Web sites for informational purposes. Wind River accepts no responsibility for the information provided in such third-party documentation.

Corporate Headquarters
Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): (800) 545-WIND
telephone: (510) 748-4100
facsimile: (510) 749-2010

For additional contact information, please visit the Wind River URL:

http://www.windriver.com

For information on how to contact Customer Support, please visit the following URL:

http://www.windriver.com/support

Wind River Workbench User's Guide, 3.0 (Linux Version)
Edition 2
18 Jan 08
Part #: DOC-16056-ND-01

Contents

PART I: INTRODUCTION

1 Overview 3

1.1 Introduction 3

1.1.1 Introducing this Guide 4

1.1.2 Introducing Wind River Workbench 5

1.2 Starting Workbench 6

1.2.1 Starting Workbench on a Linux Host 6

1.2.2 Starting Workbench on a Windows Host 6

1.2.3 Deciding which Workspace to Use 7

1.3 Recognizing Concepts in Common with Eclipse 7

Windows 8
Views 9
Tabbed Notebooks 9
Moving and Maximizing Views 9
Editors 10
Perspectives 10

1.4 Understanding Cross-Development Concepts 11

Hardware in a Cross-Development Environment 12
Working on the Host 12

Connecting the Target to the Host 12

1.5 Building Different Types of Linux Projects 12

1.5.1 User Defined Projects 12

1.5.2 Managed Projects 13

1.6 Debugging in User Mode or Kernel Mode 14

1.6.1 User Mode 14

1.6.2 Kernel Mode 14

1.7 What's New in Eclipse-Related Changes 15

1.7.1 Working with Projects 15

1.7.2 Editing Source Files 16

1.7.3 Using the Outline View 17

1.7.4 Source Analysis and Symbol Browsing 17

Workbench Parser Is Now the CDT Indexer 17
Debug and Static Analysis Symbol Browsing Have Been Separated 18

1.7.5 Connecting to Targets 20

1.7.6 Working with Debugging Views 20

1.7.7 For More Information 25

1.8 Finding Related Documentation 25

1.8.1 Wind River Workbench Online Help 25

1.8.2 Wind River Workbench for Linux Documentation 25

PART II: GETTING STARTED

2 Introducing Application Development 31

2.1 Introduction 31

2.2 Running Hello World in a Native-Mode Linux Environment 32

2.2.1 Building the Hello World Application 32

2.2.2 Debugging hello_world with the Wind River Debugger 33
2.2.3 Debugging Self-Hosted Applications with the CDT/GDB Debugger 35

2.3 Creating a Cross-Debug Mode Project 37
2.3.1 Checking and Setting Your Build Specs 37
2.3.2 Creating a Wind River Linux Application Project 38
2.3.3 Importing Existing Source Files 38
2.3.4 Importing Build Specs 39

2.4 Using the Editor 40
2.4.1 Changing File Preferences 40
2.4.2 Navigating in the Source 41
Using the Outline View 41
Finding Elements 41
Finding Strings 42
2.4.3 Using Code Completion to Suggest Elements 42
2.4.4 Getting Parameter Hints for Routine Data Types 42
2.4.5 Finding Beginning and End of a Code Section 43
2.4.6 Using Bookmarks 43
Introducing an Error for this Tutorial 44
Creating the Bookmark to the Error 44
Locating and Viewing the Bookmark 44

2.5 Configuring Project Build Properties 45

2.6 Building the Project and Correcting Simple Errors 46
2.6.1 Building a Project with Introduced Errors 46
2.6.2 Displaying a File's History 47
2.6.3 Rebuilding the Project 47

3 Creating and Debugging Platform Projects 49

3.1 Introduction 50
3.1.1 Checklist for Creating a Platform Project to Debug 51

3.1.2 Exploring Targets and Making Easier Connections 53
3.1.3 Debugging Applications 53
3.1.4 Creating External Projects and Using Layers and Templates 54

3.2 Creating a Wind River Linux Platform Project 54

3.3 Building the Target's File System and Kernel 57

3.3.1 Building the File System 57
3.3.2 Building the Kernel 57
3.3.3 Understanding What Else You Can Build 58

3.4 Configuring Target Connections 61

3.4.1 Creating and Configuring QEMU Connections 61
3.4.2 Configuring Hardware Target Connections 63
Configuring NFS to Export Root File Systems to the Target 63
Configuring TFTP to Provide a Kernel 64
Creating an Actual Target Connection 64

3.5 Running Usermode Agent on Non-Emulated Targets 65

3.6 Testing Target Connections 67

3.7 Using the Remote System Explorer 67

Browsing the Target's File System 68
Copying Files Within the Target File System 68
Issuing Remote Shell Commands to Your Target 68

3.8 Using the One-Step Launch Configuration 69

3.8.1 Accepting the Automatic Debug Launch Configuration 69
3.8.2 Modifying the Default Launch Configuration 70
3.8.3 Using the Launch Configuration to Connect to the Target 71

3.9 Debugging an Application on the Target 72

3.9.1 The Workbench Debugging Environment 72
3.9.2 Running on the Target up to a Breakpoint 73

3.9.3 Using the Device Debug Perspective 74
3.9.4 Stepping Through Code 76
3.9.5 Setting and Running to a Breakpoint 78
3.9.6 Modifying the Breakpoint to Execute Continuously 80
3.9.7 Sending Signals to Processes 80
3.9.8 Terminating the Process 81

3.10 Customizing Builds with Layers and Templates 81
3.10.1 Adding Templates to a Project 82
3.10.2 Adding Layers to a Project 83

3.11 Working with External Locations 84
3.11.1 Creating User-Defined Projects at External Locations 84
3.11.2 Building usermode-agent at External Locations 85

4 Kernel Debugging (Kernel Mode) 87

4.1 Introduction 88

4.2 Background Steps and Concepts 89
4.2.1 Setting Up for kgdboe Between a Target and a Development Host . 89
4.2.2 Using agent-proxy to Cross Subnet Boundaries for kgdboe 89
4.2.3 Mounting the Target's File System via NFS 91
4.2.4 Kernels: Enabling KGDB versus Debug Builds 92
4.2.5 KGDB and Safe Areas when Debugging the Kernel 92

4.3 Preparing the Target for KGDB 93
4.3.1 Option 1: Using a UDP Connection on a Non-PCD Target 93
4.3.2 Option 2: Using a UDP Connection on a PCD Product 94
Statically Configuring KGDB-OE for PCD (uclibc_small+small) 94
Dynamically Configuring KGDB-OE for PCD (uclibc_small+small) 94
4.3.3 Option 3: Using a Serial/Telnet Connection 95

4.4 Connecting with KGDB from Workbench 95
4.4.1 Option 1: Making an Ethernet Connection 95
4.4.2 Option 2: Making a Telnet Connection 97
4.4.3 Option 3: Making a Serial Cable Connection 98
4.4.4 When a KGDB Connection Fails 98

4.5 Attaching and Debugging the Kernel from Workbench 99
4.5.1 If the Compilation Unit Source Is Not Found 100
4.5.2 If the Target's Terminal Window Is Stopped 100
4.5.3 Setting and Removing a Sample Breakpoint 100

4.6 Debugging User-Supplied Kernel Modules 101
4.6.1 Building the Sample User-Supplied Kernel Module 101
4.6.2 Debugging a Running User-Supplied Kernel Module 102
4.6.3 Stepping into User-Supplied Kernel Module init from module.c 104
4.6.4 Placing Breakpoints in Modules Not Yet Loaded 106
4.6.5 Providing Object Path Mappings to All Modules 107
4.6.6 Using the testmod Enhanced Sample Kernel Module 108
4.6.7 Creating a Custom Kernel Module 109

4.7 Using QEMU for KGDB Kernel Debugging 110

4.8 Enabling and Disabling KGDB in the Kernel 113

5 Configuring Wind River Linux Platforms 115

5.1 Introduction 115

5.2 Configuring Wind River Linux Platform Kernels 116
5.2.1 Kernel Configuration Node 116

5.3 Adding Kernel Modules to the Platform 117
5.3.1 Creating a Custom Kernel Module 118
5.3.2 Moving Kernel Module Projects 119

5.4 Adding Build Targets for Analysis Tools 120

5.5 Working with Packages and Target File Systems 121

5.5.1 Introducing the User Space Configuration Node 121

5.5.2 Exploring the Target File System 121

5.5.3 Handling Target File System Packages (RPMs) 123

Working with the Controls for Packages 124
Force Installing Packages 125
Removing Packages 125
Installing Packages 126
Installing Multi-lib Packages and Exporting Their Sysroots 126
Importing Packages 127
Building Packages 127
Debugging Packages 128

5.5.4 Using the File System Layout Configuration Tool 129

Using the Context Menu Entries 129
Examining Meta Properties of the Files in the File System 131
Adding Files and Directories to the File System 131
Working with Packages 132
Adding a Device and Changing Ownership 132

5.6 Using Quilt to Manage Patches 133

5.6.1 Setting Up Quilt to Manage Patches 133

5.6.2 Using the Quilt Commands 134

5.6.3 Using Quilt to View Patch Annotations 135

5.6.4 Using Quilt to Import Source and SRPM Packages 136

5.6.5 Exporting and Importing Patches with Workbench 137

5.7 Automating Target Deployment 138

PART III: PROJECTS

6 Projects Overview 143

6.1 Introduction 143

6.2 Workspace and Project Location 144

6.3 Creating New Projects 145

6.3.1 Subsequent Modification of Project Creation Wizard Settings 146

6.3.2 Projects and Application Code 146

6.4 Overview of Preconfigured Project Types 146

Wind River Linux Application Project 147
Wind River User-Defined Project 147
Wind River Linux Platform Project 147
Wind River Linux Kernel Module Project 147
Customer Specific Linux Application Project 147
Customer Specific Linux Kernel Project 148
User-Defined Project 148
Native Application Project 148

6.5 Projects and Project Structures 148

6.5.1 Adding Subprojects to a Project 149

6.5.2 Removing Subprojects 149

7 Creating User-Defined Projects 151

7.1 Introduction 151

7.2 Creating and Maintaining Makefiles 152

7.3 Creating User-Defined Projects 152

7.4 Configuring User-Defined Projects 153

7.4.1 Configuring Build Support 153

7.4.2 Configuring Build Targets 154

8 Creating Native Application Projects 155

8.1 Introduction 155

8.2 Creating a Native Application Project 156

8.3 Application Code for a Native Application Project 158

PART IV: DEVELOPMENT

9 Working in the Project Explorer 161

9.1 Introduction 161

9.2 Creating Projects 162

9.3 Adding Application Code to Projects 162

Importing Resources 162
Adding New Files to Projects 163

9.4 Opening and Closing Projects 163

9.5 Scoping and Navigation 164

9.6 Moving, Copying, and Deleting Resources and Nodes 165

9.6.1 Understanding Resources and Logical Nodes 166

9.6.2 Manipulating Files 166

9.6.3 Manipulating Project Nodes 167

Moving and (Un-)Referencing Project Nodes 167
Deleting Project Nodes 167

9.6.4 Manipulating Target Nodes 168

10 Navigating and Editing 169

10.1 Introduction 169

10.2 Wind River Workbench Context Navigation 170

The Symbol Browser 171
The Outline View 171
The File Navigator 172
Type Hierarchy View 172
Include Browser 173

10.3 The Editor 173

Code Templates 173
Provided Templates 174
Configuring a Custom Editor 175

10.4 Search and Replace 175

10.5 Configuring the Indexer for Source Analysis 176

10.5.1 Setting Indexer Preferences 176

Global (Workspace) Preferences 176
Project-Specific Properties 177

10.5.2 Editing Build Properties 177

Setting Build Properties for Managed Projects 177
Setting Build Properties for User-Defined Projects 179

10.5.3 Setting Up Paths and Symbols 180

Managing Include Paths for the Indexer (Include Paths Tab) 181
Configuring Indexing of Symbols (Symbols Tab) 183
Configuring Sources and Exclusion Filters (Sources / Filters Tab) .. 183
Setting Up a Build-Driven Index (Discovery Tab) 184
Specifying User-Private Paths and Symbols (Miscellaneous Tab) 185
Specifying External APIs (External APIs Tab) 186

10.5.4 Updating a Project's Index 186

10.5.5 Sharing Source Analysis Data with a Team 187

11 Building Projects 189

11.1 Introduction 189

11.2 Configuring Managed Builds 191

Adding Build Targets to Managed Builds 191
Modifying Build Targets 192
Leveling Attributes 194
Understanding Managed Build Output 194

11.3 Configuring User-Defined Builds 195

11.4 Accessing Build Properties 196

11.4.1 Workbench Global Build Properties 196
11.4.2 Project-specific Build Properties 196
11.4.3 Folder, File, and Build Target Properties 196
11.4.4 Multiple Target Operating Systems and Versions 197

11.5 Makefiles 197

11.5.1 Derived File Build Support 198
The Yacc Example 198
General Approach 199

12 Building: Use Cases 201

12.1 Introduction 201

12.2 Adding Compiler Flags 202

Adding a Compiler Flag by Hand 202
Adding a Compiler Flag with GUI Assistance 203

12.3 Building Applications for Different Target Architectures 204

12.4 Creating Library Build Targets for Testing and Release 204

12.5 Implementing Architecture-Specific Functions 207

12.6 User-Defined Build Targets in the Project Explorer 210

12.6.1 Custom Build Targets in User-Defined Projects 210
12.6.2 Custom Build Targets in Workbench Managed Projects 210
12.6.3 Custom Build Targets in Wind River Linux Platform Projects 211
12.6.4 User Build Arguments 212

12.7 Developing on Remote Hosts 213

12.7.1 General Requirements 213
12.7.2 Remote Build Scenarios 214
Local Windows, Remote UNIX 214
Local UNIX, Remote UNIX 214
Local UNIX, Remote Windows 214

12.7.3 Setting Up a Remote Environment 214
12.7.4 Building Projects Remotely 215
12.7.5 Running Applications Remotely 216
12.7.6 Rlogin Connection Description 217
12.7.7 SSH Connection Description 217
12.7.8 Example Using Samba on Remote Linux Host 218
Configure the Remote Linux Host 218
Configure the Windows Host 219
Configure Workbench 220
Create an Example Project 220
Run the Application on the Remote Host 221
Debug the Application on the Remote Host 221

PART V: TARGET MANAGEMENT

13 Connecting to Targets 227

13.1 Introduction 227

13.2 The Remote Systems View 228

13.3 Defining a New Connection 228

13.4 Establishing a Connection 230

13.5 Connection Settings 230

Connection Template 231
Back End Settings 231
Target File System and Kernel 232
Advanced Options (KGDB Only) 232
Advanced Target Server Options 232
Command Line 233

13.5.1 Target Operating System Settings 233
13.5.2 Object Path Mappings 234
13.5.3 Specifying an Object File 234
Pathname Prefix Mappings 235

Basename Mappings 235
13.5.4 Target State Refresh Page 235
Available CPU(s) on Target Board 235
Initial Target State Query and Settings 236
Target State Refresh Settings 236
Listen to Execution Context Life-cycle Events 236
13.5.5 Connection Summary Page (Target Server Connection) 236
13.6 The Registry 237
13.6.1 Launching the Registry 237
13.6.2 Remote Registries 238
13.6.3 Shutting Down the Registry 239
13.6.4 Changing the Default Registry 239
14 Connecting with USB 241
14.1 Introduction 241
14.2 Configuring a Target for USB Connection 242
Target Configuration for a Linux Kernel 2.6 Host 242
Target Configuration for a Windows Host 243
14.3 Configuring a Host for USB Connection 243
Linux 2.6 Host Configuration 243
Windows Host Configuration 244
15 Connecting with TIPC 245
15.1 Introduction 245
15.2 Configuring TIPC Targets 246
15.2.1 Installing the TIPC Kernel Module 247
15.2.2 Running the usermode-agent 247
15.3 Configuring a TIPC Proxy 248

15.4 Configuring Your Workbench Host 250

15.5 Using usermode-agent with TIPC 251

PART VI: DEBUGGING

16 Managing Breakpoints 257

16.1 Introduction 257

16.2 Types of Breakpoints 258

16.2.1 Line Breakpoints 259

16.2.2 Expression Breakpoints 259

16.2.3 Hardware Breakpoints 259

Adding Hardware Instruction Breakpoints 260
Adding Hardware Data Breakpoints 260
Disabling and Removing Hardware Breakpoints 261
Converting Breakpoints to Hardware Breakpoints 261
Comparing Software and Hardware Breakpoints 261

16.3 Manipulating Breakpoints 262

16.3.1 Exporting Breakpoints 262

16.3.2 Importing Breakpoints 262

16.3.3 Refreshing Breakpoints 262

16.3.4 Disabling Breakpoints 263

16.3.5 Removing Breakpoints 263

17 Launching Programs 265

17.1 Introduction 265

17.2 Creating a Launch Configuration 266

17.2.1 Editing an Attach to Target Launch Configuration 267

The Main Tab 267
The Projects to Build Tab 267

The Source Tab 268
The Common Tab 268

17.2.2 Creating a Process Launch Configuration 269

17.2.3 The Main Tab 269

17.2.4 The Projects to Build Tab 269

17.2.5 The Debug Options Tab 270

17.2.6 The Source Tab 270

17.2.7 The Common Tab 270

17.2.8 Using Launch Configurations to Run Programs 271

Increasing the Launch History 271
Troubleshooting Launch Configurations 272

17.3 Remote Java Launches 272

17.4 Launching Programs Manually 275

17.5 Controlling Multiple Launches 276

Terminology 276
Configuring a Launch Sequence 276
Pre-Launch, Post-Launch, and Error Condition Commands 277

17.6 Launches and the Console View 281

Launches and the Console View 281
Console View Output 281

17.7 Attaching the Debugger to a Running Process 283

17.8 Attaching to the Kernel 284

17.9 Suggested Workflow 285

18 Debugging Projects 287

18.1 Introduction 287

18.2 Using the Debug View 288

18.2.1 Configuring Debug Settings for a Custom Editor 289

18.2.2 Understanding the Debug View Display 290
How the Selection in the Debug View Affects Activities 291
Monitoring Multiple Processes 291
Colored Views 293
18.2.3 Stepping Through a Program 293
18.2.4 Using Debug Modes 294
18.2.5 Setting and Recognizing the Debug Mode of a Connection 295
18.2.6 Debugging Multiple Target Connections 295
18.2.7 Disconnecting and Terminating Processes 296
18.2.8 Using the Debug View to Send Signals to Processes 296
Resuming Suspended Processes 296
Editing a Process's Signal Configuration 297
18.2.9 Changing Source Lookup Settings 297

18.3 Stepping Through Assembly Code 298

18.4 Using the Disassembly View 301
18.4.1 Opening the Disassembly View 301
18.4.2 Understanding the Disassembly View Display 301

18.5 Java-JNI Cooperative Debugging 302
Configuring a User Mode Connection for Cooperative Debugging 302
Creating a Launch Configuration for Cooperative Debugging 303
Debugging In Java and Native Modes 304
Conditions that Disable the JDT Debugger 305
Re-Enabling the JDT Debugger 305

18.6 Remote Kernel Metrics 306
Building and Running the RKM Monitor 306
Running the RKM Monitor from the Command Line 307
Attaching Data Monitor to the RKM Monitor 307
Using Data Monitor to View Remote Kernel Metrics 308

18.7 Run/Debug Preferences 308

19 Analyzing Core Files 309

19.1 Introduction 309

19.2 Acquiring Core Dump Files 310

19.3 Attaching Workbench to a Core File 311

Core File Analysis 312
Ending the Session 312

PART VII: UPDATING

20 Integrating Plug-ins 315

20.1 Introduction 315

20.2 Finding New Plug-ins 316

20.3 Incorporating New Plug-ins into Workbench 316

20.3.1 Creating a Plug-in Directory Structure 316

20.3.2 Installing a ClearCase Plug-in 317

Downloading the IBM Rational ClearCase Plug-in 318
Adding Plug-in Functionality to Workbench 319

20.4 Disabling Plug-in Functionality 320

20.5 Managing Multiple Plug-in Configurations 320

20.6 Installing JDT for Third-Party Plug-ins and Debugging 321

21 Using Workbench in an Eclipse Environment 323

21.1 Introduction 323

21.2 Recommended Software Versions and Limitations 324

Java Runtime Version 324
Eclipse Version 324
Defaults and Branding 324

21.3 Setting Up Workbench 324

21.4 Using CDT and Workbench in an Eclipse Environment 325

21.4.1 Workflow in the Project Explorer 326

Application Development Perspective (Workbench) 326
C/C++ Perspective (CDT) 326

21.4.2 Workflow in the Build Console 327

Application Development Perspective (Workbench) 327
C/C++ Perspective (CDT) 327
General 327

21.4.3 Workflow in the Editor 327

Opening Files in an Editor 327

21.4.4 Workflow for Debugging 328

Workbench and CDT Perspectives 328

22 Using Workbench with Version Control 329

22.1 Introduction 329

22.2 Using Workbench with ClearCase Views 329

22.2.1 Adding Workbench Project Files to Version Control 330

22.2.2 Choosing Not to Add Build Output Files to ClearCase 331

PART VIII: REFERENCE

A Troubleshooting 335

A.1 Introduction 335

A.2 Startup Problems 336

Workspace Metadata Is Corrupted 336
.workbench-3.0 Directory Is Corrupted 337
Registry Unreachable (Windows) 337
Workspace Cannot Be Locked (Linux and Solaris) 338
Pango Error on Linux 339

A.3 General Problems 339

A.3.1 Help System Does Not Display on Linux 339

A.3.2 Help System Does Not Display on Windows 340

A.3.3 Resetting Workbench to Its Default Settings 340

A.4 Fixing Indexer Issues 341

A.4.1 Indexing Problems with Managed Projects 341

A.4.2 Indexing Problems with User-defined Projects 341

Source Files Not Yet Built 341
Unsuccessful Built Output Analysis 342

A.4.3 Other Indexing Problems 342

Files Excluded in the GUI 342
Outdated Index 343
Incorrect Include Paths and Symbols 343
Troubles Parsing Source Code 343

A.5 Hints about Performance and Breakpoints 344

Module Optimization Levels and Jumping Program Counters 344
Module Optimization Levels and Skipped Breakpoints 345
Manual Refresh of the Build Linked Directory in Workbench 345
Disabling Workspace Refresh on Startup 345
Workbench Freezes: No Response to Mouse Clicks 346
Planting (Inserting) Breakpoints 346

A.6 Error Messages 346

A.6.1 Project System Errors 347

Project Already Exists 347
Cannot Create Project Files in Read-only Location 347

A.6.2 Build System Errors 348

A.6.3 Building Projects While Connected to a Target 348

A.6.4 Remote Systems View Errors 349

Troubleshooting Connecting to a Target 350
Exception on Attach Errors 351
Error When Running a Task Without Downloading First 351
Downloading an Output File Built with the Wrong Build Spec 351
Error if Exec Path on Target Is Incorrect 352

Troubleshooting Running a Process 353
A.6.5 Launch Configuration Errors 354
A.6.6 Source Analysis Errors 354

A.7 Error Log View 355

A.8 Error Logs Generated by Workbench 355
A.8.1 Creating a ZIP file of Logs 355
A.8.2 Eclipse Log 357
A.8.3 DFW GDB/MI Log 357
A.8.4 DFW Debug Tracing Log 358
A.8.5 Debugger Views GDB/MI Log 358
A.8.6 Debugger Views Internal Errors Log 359
A.8.7 Debugger Views Broadcast Message Debug Tracing Log 359
A.8.8 Target Server Output Log 360
A.8.9 Target Server Back End Log 361
A.8.10 Target Server WTX Log 362
A.8.11 Remote Systems View Debug Tracing Log 362

A.9 Technical Support 363

B Workbench and Command-Line Techniques 365
Workbench and the Command Shell Environment 365
Setting the Environment Without Starting a Sub-shell 368

C Installing on Network Servers 371
Shared Wind River Linux Installation 372
Local Workbench Installation 373

D Configuring a Wind River Proxy Host 375

D.1 Introduction 375

D.2 Configuring wrproxy 377

Configuring wrproxy Manually 377
Creating a wrproxy Configuration Script 378

D.3 wrproxy Command Summary 379

Invocation Commands 379
Configuration Commands 379

E Command-line Updating of Workspaces 383

Execution 383
Options 384

F Command-line Importing of Projects 387

F.1 Introduction 387

F.2 wrws_import Reference 388

Execution 388
Options 388

G Starting usermode-agent at Boot Time 391

H Glossary 393

Index 401

PART I

Introduction

1 Overview 3

1 Overview

1.1 Introduction 3

1.2 Starting Workbench 6

1.3 Recognizing Concepts in Common with Eclipse 7

1.4 Understanding Cross-Development Concepts 11

1.5 Building Different Types of Linux Projects 12

1.6 Debugging in User Mode or Kernel Mode 14

1.7 What's New in Eclipse-Related Changes 15

1.8 Finding Related Documentation 25

1.1 Introduction

Wind River Workbench is an IDE development suite that lets you develop Linux target kernels and embedded applications very efficiently.

Workbench runs on various host operating systems.

1.1.1 Introducing this Guide

This document is divided into the following parts:

Part I. Introduction (this chapter) provides an overview of documentation, introduces Eclipse and cross-development concepts, and outlines how Workbench helps cross-development

Part II. Getting Started provides an introductory tutorial that takes you from starting Workbench to using it to perform common debugging activities on sample applications in user mode. Additional chapters build on this knowledge to show how to develop Wind River Linux platform projects and introduce kernel-mode debugging.

Part III. Projects provides detailed information on how to use projects in Workbench, including pre-defined projects, user-defined projects, using the Project Explorer, and a discussion of various advanced project scenarios.

Part IV. Development describes how to use Workbench to edit source code, build projects, and parse and analyze source-code symbol information.

Part V. Target Management discusses connecting to targets, and how to create new target connections.

Part VI. Debugging provides an in-depth look at debugging operations, including launching programs, managing breakpoints, and troubleshooting.

Part VII. Updating discusses integrating plug-ins to Workbench.

Part VIII. Reference describes how to use the command line to update workspaces for automated builds and to import projects . It also provides source code for a program patch example, a glossary, and an index.

In this document, italics mark terms being introduced, as well as placeholders for values you supply. Bold indicates literal values. For example, *installDir* refers to the location in which you have installed Workbench. (By default, this is **C:\WindRiver** on Windows hosts and **$HOME/WindRiver** on Linux and Solaris hosts.)

Bold also indicates menu choices, as in **File > New > Project**, meaning to select **File**, then **New**, then **Project**. Commands that you enter on a command line are shown in bold, and system output is shown in typewriter text, for example:

```
$ pwd
/home/mary/WindRiver
$
```

The screenshots in this document were taken on a host running Red Hat Enterprise Linux, so they may differ slightly from what you see on your screen.

1.1.2 Introducing Wind River Workbench

Workbench is an integrated development environment for creating device software to run on embedded Wind River Linux systems. Workbench is optimized for both small programs and very large ones with thousands of files and millions of lines of code. It includes a full project facility, advanced source-code analysis, simultaneous management of multiple targets, and a debugger with capabilities for managing multiple processes or threads on a single target or on multiple targets.

Workbench and Wind River Linux have been developed in parallel to make it easy to develop Wind River Linux platform targets. You may use any combination of Linux command-line and Workbench GUI actions as desired.

Workbench ensures the smallest possible difference between the performance of the target you use during development and the performance of the target after deployment of the finished product by keeping development tools on the host.

With Workbench, your application does not need to be fully linked. Partially completed modules can be downloaded for incremental testing. Modules do not need to be linked with the run-time system, or even with each other. The host-resident shell and debugger can be used interactively to invoke and test either individual application routines or complete tasks.

Workbench loads the relocatable object modules directly and maintains a complete host-resident symbol table for the target. This symbol table is incremental: the target server incorporates symbols as it downloads each object module. You can examine variables, call routines, spawn tasks, disassemble code in memory, set breakpoints, trace subroutine calls, and so on, all using the original symbol names.

Workbench shortens the cycle between developing an idea and implementing it by allowing you to quickly download your incremental run-time code and dynamically link it with the operating system. Your application is available for symbolic interaction and testing with minimal delay.

The Workbench debugger allows you to view and debug applications in the original source code. Setting breakpoints, single-stepping, examining structures, and so on, are all done at the source level, using a convenient graphical interface.

1.2 Starting Workbench

You can run Workbench on a Linux or Windows host. On a Windows host, you have fewer options related to the target.

1.2.1 Starting Workbench on a Linux Host

You can find the shell command to start Workbench in *installDir*.

To start Workbench:

1. Make sure your path environment variable is set to include the path to your compiler. Typically, **which gcc** should yield **/usr/bin/gcc**.
2. From your Workbench *installDir*, issue the following command:

   ```
   $ ./startWorkbench.sh
   ```

   This is the basic startup command. You can supply arguments as described in *Running Eclipse* in *Eclipse Workbench User Guide:Tasks* in the online help. For example, these arguments give more heap space:

   ```
   $ ./startWorkbench.sh -vmargs -Xmx512m
   ```

   The resulting Welcome screen lets you click on options to see an overview, find tutorials, see what is new, and to run Workbench. To come back to this screen, select **Help > Welcome** from the Workbench window.
3. Click **Workbench**.

   Workbench displays the **Application Development** perspective. Figure 1-1 shows the default arrangement of the views. Workbench resumes the same perspective when you open it again, or you can select the default settings by choosing **Window > Reset Perspective**.

1.2.2 Starting Workbench on a Windows Host

To start Workbench on a Windows host:

1. From the **Start** menu, select **All Programs > Wind River > Workbench 3.*x* > Wind River Workbench 3.*x***.

2. Continue reading this chapter and following the step-by-step instructions in the next chapter until it is time to connect to the local target, at which point, you should skip ahead to *2.3 Creating a Cross-Debug Mode Project*, p.37.

1.2.3 Deciding which Workspace to Use

Workbench uses a *workspace* to hold different types of information, including:

- Information about a set of projects, including project names, lists of files for each project, and build specifications.
- Information about the current session, including the types and positions of your windows when you last exited Workbench, current projects, and installed breakpoints.

When you start Workbench, you need to specify a workspace to store your project information. You must have write access to the workspace you are going to use.

The default location proposed by Wind River Workbench may be sufficient, but:

- We recommend you separate your workspace from the installation directory.
- If your target and host share a root file system, consider placing your workspace inside the root file system that you export to the target.
- Plan to keep different sets of projects separate: each workspace has its own set of projects.
- Consider using two or more instances of Workbench, where each must have its own workspace and its own target root file system.

See *3.4.2 Configuring Hardware Target Connections*, p.63, for more information.

Note that you can switch from your current workspace to another by selecting **File > Switch Workspace.**

1.3 Recognizing Concepts in Common with Eclipse

Wind River Workbench is based on the Eclipse platform. Basic features of Workbench are common to all Eclipse-based systems. For a more details, select **Help > Help Contents > Wind River Partners Documentation**, then expand the

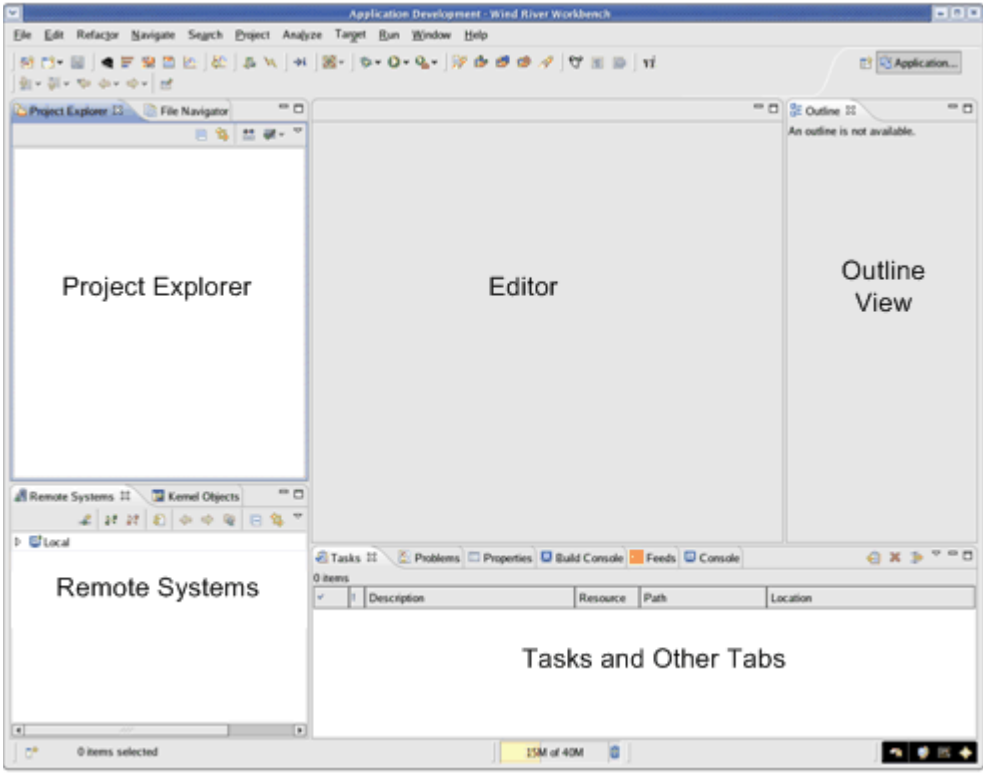
Eclipse Workbench User Guide entry) or navigate to the Eclipse *Workbench Users Guide* at **www.eclipse.org/documentation/main.html**.

Window, *view*, *tabbed notebook*, *editor*, and *perspective* describe features of Eclipse-based systems. This section outlines these terms and explains how to use these features.

Windows

The term *window* is used only for the overall outer frame shown in Figure 1-1. Use **Window > New** to create a new outer frame as part of the same Workbench session; then use the new window to open additional views within the same perspective (see *Perspectives*, p.10).

Figure 1-1 **Workbench Windows**



Views

The term *view* refers to the individual panes within a window; in Figure 1-1, these include the Project Explorer view on the top-left side of the screen, the Outline view on the top-right, the Remote Systems view on the bottom-left, and the stacked view on the bottom-right with the title **Tasks**.

There are two rules to consider when using views:

1. Only one view (such as the Editor view—see *Editors*, p.10) can be active at a time. The title bar of the active view is highlighted.
2. Only one instance of a type of view can be present in a perspective at a time.

NOTE: Multiple editors can be present to view multiple source files.

Many views include a menu that you access by clicking the down arrow to the right of the title bar. This menu typically contains items that apply to the entire contents of the view rather than a selected item within the view.

To open a view and add it to the existing perspective (see *Perspectives*, p.10), select **Window > Show View**. Select the desired view from the list, or select **Other** to display expandable lists with more choices. The view is added at its default location in the window, and you can move it if you want—see *Moving and Maximizing Views*, p.9.

Tabbed Notebooks

Several views can be stacked together in a *tabbed notebook* (often the result of opening additional views). For example, the **Tasks** view at the bottom-right of Figure 1-1 has six tabs along the top. Click a tab to display its view.

Moving and Maximizing Views

Move a view by clicking either its title bar or its tab in a stacked notebook, and dragging it to a new location.

You can relocate a view by:

- Dragging it to the edge of another view, then dropping it. The area is then split, and both views are tiled in the same area. The cursor changes to an appropriate directional arrow as you approach the edge of a view.

- Dragging it to the title bar of an existing view, then dropping it. The view will be added to a stacked notebook with tabs. When you drag the view to stack it, the cursor changes to an icon of a set of stacked folders.
- Dragging it over a tab in an existing view, causing the view to be stacked in that notebook with its tab at the left of the existing view. You can also drag an existing tab to the right of another tab to arrange tabs to your liking.

To quickly maximize a view to fill the entire perspective area, double-click its title bar. Double-click the title bar again to restore it.

Editors

An *editor* is a special type of view used to edit files. You can associate different editors with different types of files such as C, C++, Assembler, and makefiles. When you open a file, the associated editor opens in the perspective's Editor area.

Any number of editors can be open at once, but only one can be active at a time. By default, editors are stacked in the Editor area, but you can tile them in order to view source files simultaneously.

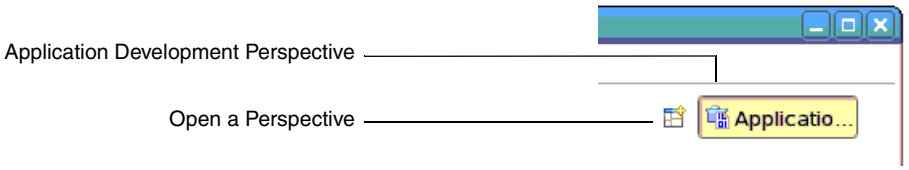
Tabs in the Editor area indicate the names of files that are currently open for editing. An asterisk (*) indicates that an editor contains unsaved changes.

Perspectives

A *perspective* is the layout of the views and editors in a window. A single window can maintain several perspectives, but shows only one perspective at a time.

To open new perspectives and switch between them, use the buttons in the shortcut bar along the top right edge of the Workbench window. When you start Workbench for the first time, two buttons display as shown in Figure 1-2.

Figure 1-2 **Perspectives Shortcut Bar**



Perspectives are a convenience for accomplishing specific tasks. For example, the Application Development perspective is designed for creating projects, browsing files, editing source code, and building applications.

For the Device Debug perspective, useful when running and debugging a program, click **Window > Open Perspective > Device Debug** This perspective contains the Debug and Breakpoints views, and a tabbed notebook with view such as Variables and Expressions. (The symbol >> indicates additional tabs that do not fit on the view. The tabs include Expressions, History, Memory, Registers, and Variables.)

You can also create your own perspectives:

1. Arrange the views in the window by opening any required views and moving them to an appropriate location.
2. Select **Window > Save Perspective As**, and enter a name for your custom perspective.
3. Click **OK**.

Workbench saves your customized perspective in your workspace. The button in the shortcut bar displays the new name when you hover the cursor over it.

Switch back to the Application Development perspective by selecting it on the top right side and resetting it with **Window > Reset Perspective**. It should once again appear as shown in Figure 1-1.

1.4 Understanding Cross-Development Concepts

Cross-development is the process of writing code on one system, known as the host, that will run on another system, known as the target.

Cross-development allows you to write code on a system that is readily available and familiar (such as a PC running Linux or Windows) and produce applications that run on hardware that you would have no other convenient way of programming, such as a chip destined for a mobile phone.

Hardware in a Cross-Development Environment

A typical host is equipped with large amounts of RAM and disk space, backup media, printers, and other peripherals. In contrast, a typical target has only the resources required by the real-time application and perhaps some small amount of additional resources for testing and debugging.

Working on the Host

You use the host as if you were writing code to run it. You can manage project files to edit, compile, link, and store multiple versions of your real-time application code. You can also configure the operating system that will run on the target.

Connecting the Target to the Host

A number of alternatives exist for connecting the target system to the host, but usually the connection consists of an Ethernet or serial link, or both.

1.5 Building Different Types of Linux Projects

You can use Workbench to create platform (kernel and file system) projects and application projects for Wind River Linux Platforms.

Workbench has predefined project types for building Wind River Linux platforms and applications. Additional project types are user-defined applications and native applications. Some of the major project types available to you are described in this section.

1.5.1 User Defined Projects

User defined projects are typically ones where the source code, including makefile, is provided by you or a third party. You import or reference the source and use the makefile provided to build it. Workbench provides the Project Explorer as a convenient way to access your build environment, and the Build Console will display build output.

1.5.2 Managed Projects

Managed projects are those for which Workbench supplies much of the tooling such as makefiles and other parts of the build system for the projects.

Wind River Linux Application Projects

A Wind River Linux Application project is a managed build, meaning Workbench supplies the makefile, determines build order, and provides macros, build paths, and build specifications (*build specs*). The build specs provided are based on the targets that are tested and supported by Wind River Linux. Appropriate libraries, cross-development tools and so on are also provided, so much of the traditional gathering and debugging of a working environment is eliminated.

Wind River Linux Platform Projects

Wind River Linux Platform projects are designed for building the entire target platform, including the kernel and file system. They provide special tools to make platform configuration, modification, and maintenance easier. Root file systems are easily built using default settings or you can configure the file system through a convenient GUI configuration tool that the project provides. Default kernels are provided for supported targets, and kernel configuration is also easily performed with a provided GUI configuration tool.

Wind River Linux Kernel Module Projects

Wind River Linux Kernel Module projects provide an easy way to create kernel modules for Wind River Linux Platforms. You can associate the kernel module with a platform and the kernel module automatically inherits the build environment of the platform.

Native Application Projects

Native application projects are managed projects that are developed and deployed on the host, so no cross-development is required. The build specs assume local development tools. Additional build specs support team development for when a team works with a combination of Windows, Linux, and Solaris host environments.

1.6 Debugging in User Mode or Kernel Mode

Depending on your needs and the version of the Linux kernel you are running on your target, you will use one or more of the following debug modes:

1.6.1 User Mode

User mode debugging allows you to perform source code debugging of user mode applications, including multi-process and multi-thread debugging. In user mode, a usermode agent on the target communicates with Workbench on the host, where you can edit your source code, step through the application, set breakpoints, and so on. Refer to *2. Introducing Application Development* for a tutorial on the use of Workbench in user mode.

The following debuggers are available for user-mode applications:

- The Wind River Workbench debugger for native-mode (self-hosted) *and* cross-developed applications
- The Eclipse CDT (C/C++ Development Toolkit) debugger for only Linux native-mode (self-hosted) development

1.6.2 Kernel Mode

Kernel mode debug allows for source code debugging of Linux kernels version 2.6 and later. The kernel must be patched for the Kernel GNU Debugger (KGDB), which communicates with the standard GNU debugger (GDB) on the host. (KGDB is enabled by default for Wind River Linux kernels and only needs to be disabled when going to production.) Kernel mode source code debugging allows you to debug the kernel using suspends, breakpoints, and so on, as you would with user mode debugging. Kernel mode debugging is described in *4. Kernel Debugging (Kernel Mode)*.

NOTE: If user mode and KGDB debugging connections are used in parallel for the same target, the user mode connection stalls when a KGDB breakpoint is reached.

1.7 What's New in Eclipse-Related Changes

Workbench has adopted the latest versions of:

- CDT—The Eclipse C/C++ Development Toolkit
- Eclipse device debugging
- Eclipse target management connections

The workflows and target connections are very similar to past versions of Workbench. But, some Eclipse views have totally replaced their Workbench counterparts; static symbol browsing has changed significantly; and, though comparable, icons and graphics are different.

This section alerts you to these and similar Eclipse-related changes. For information about other enhancements to Workbench, see the release notes for your platform at **http://www.windriver.com/support/**.

1.7.1 Working with Projects

The Eclipse Project Explorer replaces the Workbench Project Navigator. It supports your Linux application and platform projects, and also displays projects from other CDT providers. In particular, you might notice:

- Right-clicking a binary filename under the **Binaries** node allows you to launch and/or debug the binary file.
- Double-clicking parses a binary file and displays the result in the Editor.
- Double-clicking a file under the **Includes** node displays header files.
- The **User Build Arguments** field has moved to the Build Console toolbar.

You can still open the Project Navigator: select **Window > Show View > Project Navigator** from the Workbench toolbar.

Other project-related issues include:

- You cannot drag and drop changes to project hierarchies and build target contents. Instead, select **Project References > Add as Project Reference** to change relationships between projects.
- You cannot drag images from the Project Explorer to target connections.
- New data structures underlie the project system. You can import projects created with a previous version of Workbench, and they are automatically

updated to this new structure, but you will not be able to open them in a previous version of Workbench. (Migration is forward only.)

For more information about the Project Explorer, see *9. Working in the Project Explorer*; also the *C/C++ Development User Guide*, available from the Workbench help system.

1.7.2 Editing Source Files

The Workbench Editor has been replaced by the C/C++ Editor, which has many of the same features such as code folding, code completion, parameter hinting, and symbol highlighting.

New Editor features include:

- Quick access via **Ctrl-O** to the Outline view
- Auto-closing of brackets
- Indentation correction by selecting source lines and pressing **Ctrl-I**

The Editor context menu contains many of these entries, as well as many others including one for creating breakpoints directly in the Editor.

Other Editor issues include:

- A change in syntax coloring, similar to the Java editor. Select **C/C++ > Editor > Syntax Coloring** to configure other preferences.
- Different, language-dependent editors, for C/C++, Assembly, and Makefiles.
- No separate editor for Ada. Debugging still works, but you must use the default text editor for Ada files.
- ASM symbols no longer appear in the Outline view when you edit an assembly file.

For more information about the Editor, see *2.4 Using the Editor*, p.40; as well as the *Eclipse Workbench User Guide* and the *C/C++ Development User Guide*, both available from the Workbench help system.

1.7.3 Using the Outline View

The Workbench Outline view has been replaced by the Eclipse Outline view, which has the same functionality but different icons. For more information about the Outline view, see *Using the Outline View*, p.41.

1.7.4 Source Analysis and Symbol Browsing

Workbench has adopted the CDT source code indexing and symbol browsing features to replace those that appeared in previous releases.

Workbench Parser Is Now the CDT Indexer

Workbench source code parsing is now done by the CDT Indexer, which is comparable in speed to the Workbench parser it replaces.

Parsing Build Output

The Indexer provides build output parsing. If you uncheck **Enable project specific settings**, Workbench scans the output of the build to set up sources for indexing. Only those sources that are part of the build process are indexed, and all flags (defines, includes) used by the build are used for the indexing. You can manually initiate the re-indexing of files in a project, updating only those files that were modified, or all files. Updating of modified files happens automatically when they are saved, but you can also initiate re-indexing when a header file is touched.

Setting Indexer Preferences

You can configure indexer-specific exclusion filters using regular expressions. Select a project, choose **Properties > C/C++ General > Paths and Symbols > Source/Filters**, then edit the filter data and add a regular expression.

Indexer performance is affected by preference settings, such as turning off indexing of type references to speed up the index (you can still navigate typedefs, but cannot search for C/C++ references).

Sharing Symbol Data with a Team

When you import a project from a team member's different workspace, the shared indexing data is copied along with it. After the import, the data is treated as if it had been indexed by your workspace. The import itself is automatic if you use **Export > C/C++ > Team shared index** to export the data.

Indexer Issues

Other Indexer issues include:

- You cannot configure external APIs as they are now associated with a project depending on the project type. External APIs are supplied for Wind River-specific project types (such as Linux project types).
- Macro references do not appear in the call hierarchy (for MACROs that look like functions); read/write flags for variables do not appear in the Call Tree; and polymorphic method calls are not honored in the Call Tree.
- References to constructors, destructors, or implicit type conversions do not appear in the symbol list, however, declarations and definitions do.
- An implicit conversion (sequence) is generated when the type of an argument in a function call does not match the type of the parameter. It can make use of constructors to convert the argument to an object of the expected type.
- Implicit constructor calls may be needed for the initialization of base classes, automatic variables, or returned values. An implicit destructor call is necessary when an automatic variable goes out of scope.

For more information about the CDT Indexer, see *10.5 Configuring the Indexer for Source Analysis*, p.176; and the *C/C++ Development User Guide*, available from the Workbench help system.

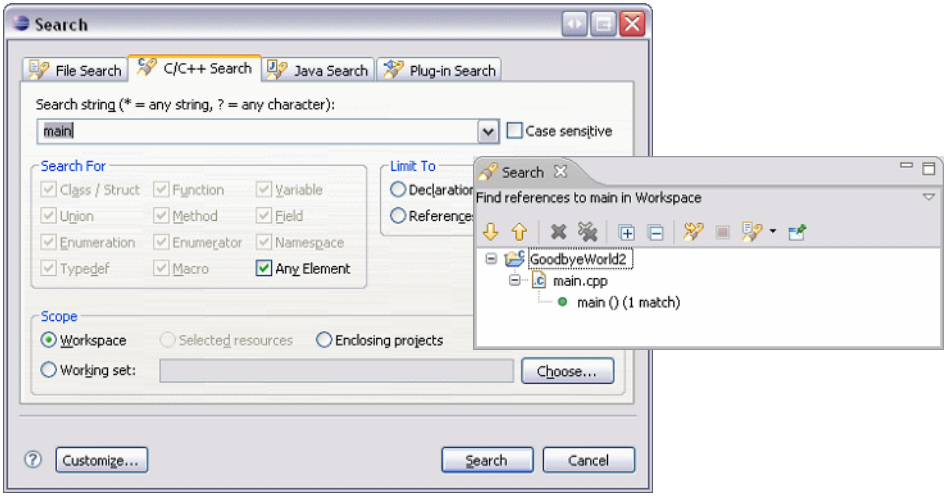
Debug and Static Analysis Symbol Browsing Have Been Separated

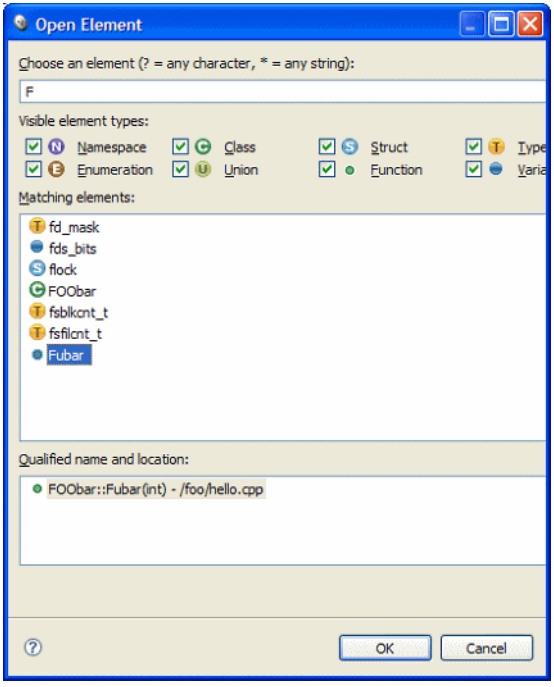
Static analysis is now known as *source analysis*.

The Symbol Browser use to display static analysis and debug symbols. Now:

- The Debug Symbol Browser replaces the Symbol Browser. They look the same but the **Debug mode** icon is gone.
- The functionality of the Static Symbol Browser has been replaced by:
  - The C/C++ Search dialog (**CTRL+H**, then select the **C/C++ Search** tab) lists results in a separate view from which you can navigate.
  - The Open Element dialog (**SHIFT+F3**) lists symbol matches as you begin typing, and you can navigate to the desired symbol by clicking it.

New C/C++ Search and Open Element Dialogs





1.7.5 Connecting to Targets

The Eclipse Remote Systems view replaces the Target Manager, offering some new features and connection types but otherwise the same functionality. The Remote Systems view allows you to configure, access, and manage remote file systems and targets, and it provides a **Local** node for access to the local file system and shell.

Previous connections that have been retained:

- Wind River Linux Application Core Dump Target Server Connection
- Wind River Linux KGDB Connection
- Wind River Linux Target Server Connection for Linux User Mode—renamed to Wind River Linux User Mode Target Server Connection
- Wind River Linux Target Server Connection for Linux—renamed to Wind River Linux Host Development Target Server Connection

New connections:

- Wind River Generic GDB Remote Serial Protocol Connection
- Wind River QEMU Backend Connection
- Wind River QEMU Connection
- Wind River QEMU Linux User Mode Target Server Connection

Target Management Issues

You cannot drag images from the Project Explorer to Wind River target connections.

Using the Remote Systems Perspective

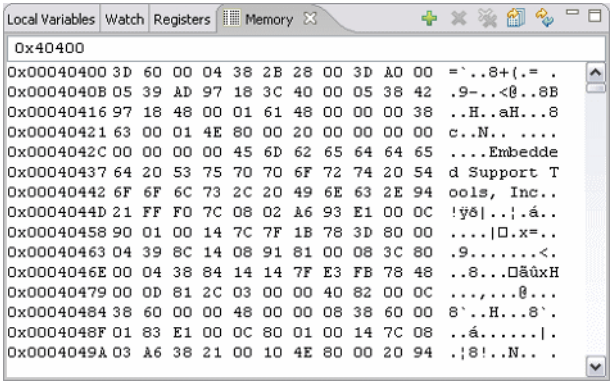
Switching to the Remote Systems perspective displays the Remote Systems Details view, which provides additional information about each defined target connection.

For more information about target management, see *13. Connecting to Targets* and the *RSE User Guide*, available from the Workbench help system.

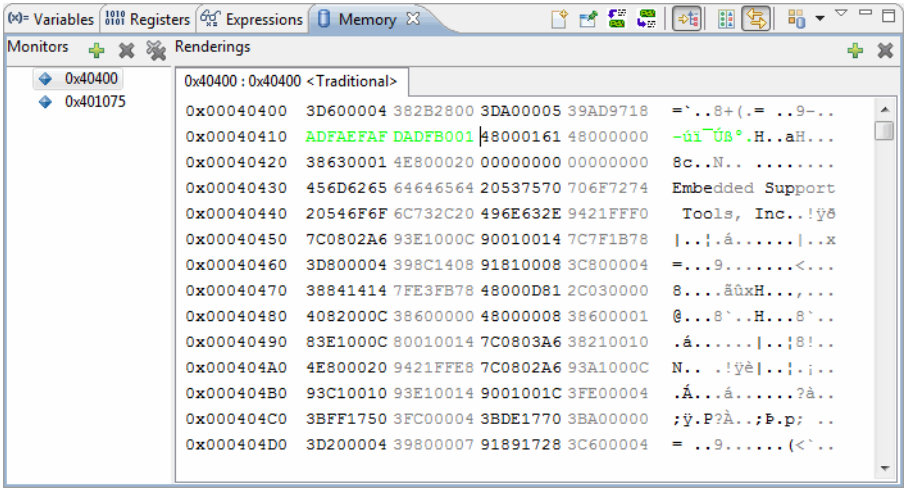
1.7.6 Working with Debugging Views

Several debugging views have been replaced by their Eclipse counterparts.

Old Workbench Memory View



New Eclipse Memory View



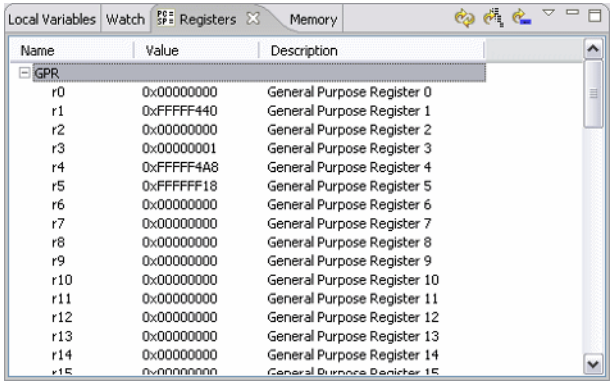
The Eclipse Memory View Replaces the Workbench Memory View

The new view, for example, replaces tabs with monitors. Renderings (one of which is installed) change how you look at memory. Find and Replace are similar; so is the delay for searches over large memory ranges. SNF import and export are no longer supported, and there is no S-record format for 64-bit import and export. Drag and drop to the Memory view is not supported. See the *C/C++ Development User Guide*, available from the Workbench help system.

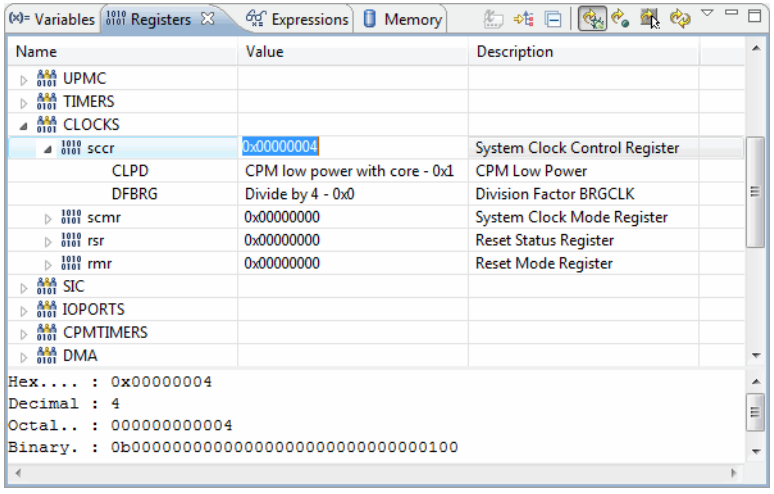
Workbench Registers View is Replaced by the Eclipse Registers View

The Eclipse Registers view now contains On-Chip Debugging (OCD) extensions. The Properties view was removed because bit fields are now editable in place in the Registers view. The Details pane (at bottom of view) displays the register value in each radix.

Old Workbench Registers View


Local Variables
Watch
Registers
Memory
Name
Value
Description
GPR
r0 0x00000000 General Purpose Register 0
r1 0xFFFFF440 General Purpose Register 1
r2 0x00000000 General Purpose Register 2
r3 0x00000001 General Purpose Register 3
r4 0xFFFFF4A8 General Purpose Register 4
r5 0xFFFFFF18 General Purpose Register 5
r6 0x00000000 General Purpose Register 6
r7 0x00000000 General Purpose Register 7
r8 0x00000000 General Purpose Register 8
r9 0x00000000 General Purpose Register 9
r10 0x00000000 General Purpose Register 10
r11 0x00000000 General Purpose Register 11
r12 0x00000000 General Purpose Register 12
r13 0x00000000 General Purpose Register 13
r14 0x00000000 General Purpose Register 14

New CDT Registers View

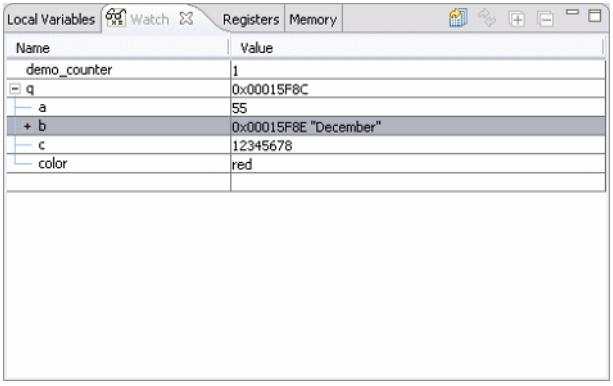


For more information about the Registers view, see *Wind River Workbench for On-Chip Debugging User Tutorials* and the *C/C++ Development User Guide*, available from the Workbench help system.

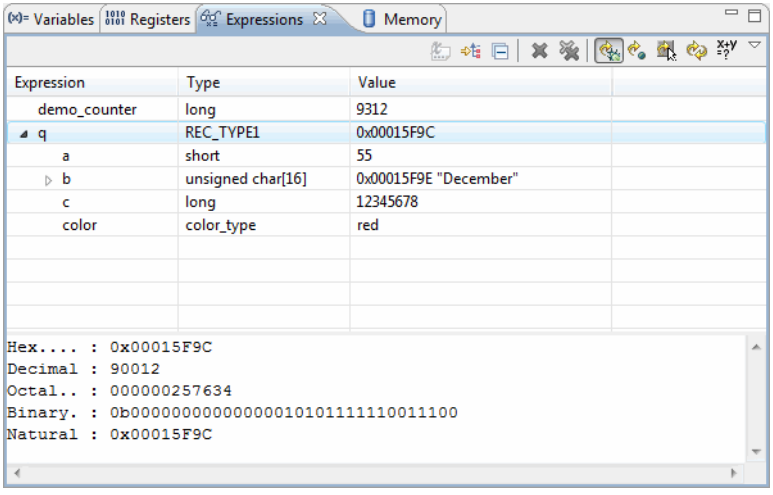
The Eclipse Expressions View Replaces the Watch View

The functionality of the two views is basically the same, and drag and drop to this view is supported. Casting pointers is supported, though you cannot set radix for a single element. The **Expand All** capability is no longer available.

Old Workbench Watch View



New Eclipse Expressions View

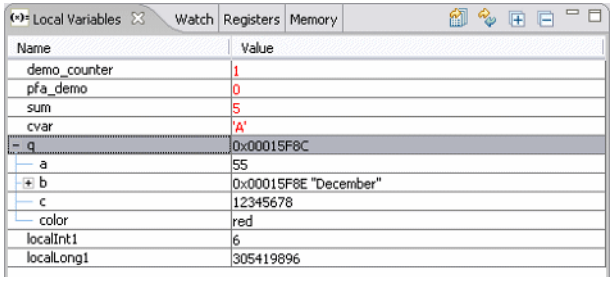


For more information about the Expressions view, see the *C/C++ Development User Guide*, available from the Workbench help system.

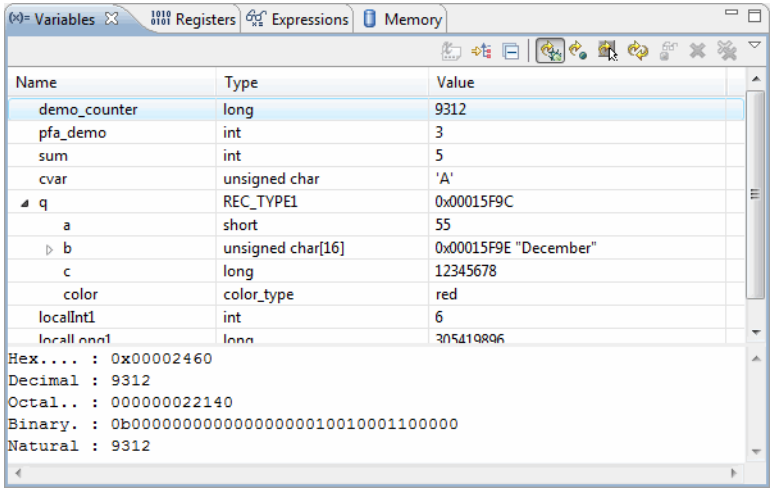
The Eclipse Variables View Replaces the Local Variables View

The functionality is basically the same, but, as in the Expressions view, you cannot set radix for a single element and **Expand All** capability is no longer available.

Old Workbench Local Variables View



New CDT Variables View



For more information about the Variables view, see the *C/C++ Development User Guide*, available from the Workbench help system.

1.7.7 For More Information

For details about new Workbench features, including non-Eclipse enhancements to Workbench, see the release notes at **http://www.windriver.com/support/**).

For more information about Eclipse and the projects introduced here, see the Eclipse documentation available from the Workbench help system, as well as documentation at **http://www.eclipse.org/documentation/**.

1.8 Finding Related Documentation

Other documentation includes:

- Wind River Workbench Online Help
- Wind River Workbench for Linux Documentation

1.8.1 Wind River Workbench Online Help

Wind River Workbench provides context-sensitive help. To access the full help set, select **Help > Help Contents in Wind River Workbench**. To see help information for a particular view or dialog box, press the help key when in that view or dialog box:

- The help key on Windows hosts is **F1**.
- On Linux and Solaris hosts, it is the combination **CTRL+F1**.

1.8.2 Wind River Workbench for Linux Documentation

The following documentation is provided online for the Workbench version that supports Wind River Linux. Additional documentation is also available online depending on your installation.

Help > Contents > Wind River Documentation > Getting Started

Wind River Workbench Tutorial, Linux Version

The Getting Started (Part II) section of this document: developing user mode applications, configuring Wind River Linux platforms, kernel mode debugging, and using the QEMU target emulator.

Help > Contents > Wind River Documentation > Guides > Host Tools

Wind River Workbench User's Guide (this document)

This guide describes how to configure your Workbench host and a Wind River Linux target to debug applications and kernel objects on the target. It describes how to use Workbench to develop projects, manage targets, and edit, compile, and debug code.

Wind River Workbench Host Shell User's Guide

The host shell is a host-resident shell that provides a command line interface for debugging targets.

Wind River Analysis Tools

This set of documents describes how to use the analysis tools that are provided with Workbench. The tools include a memory use analyzer, an execution performance profiler, a code coverage analyzer, and a graphical application variable monitoring tool.

Wind River System Viewer User's Guide

This guide describes how to use System Viewer, which is included with Workbench. System Viewer is a logic analyzer for visualizing and troubleshooting complex embedded software. The *Wind River System Viewer API Reference* is also included.

Wind River System Linux Platform User's Guide

The Wind River Linux Platforms development environment provides an efficient way to develop Linux target kernels and embedded applications. Development environments are available on a number of host platforms, and support a large and ever-growing set of targets.

Help > Contents > Wind River Documentation > References > Host Tools

Wind River Workbench User Interface Reference

This provides specific reference material for the Workbench GUI: detailed information on individual menu choices, buttons, and so on, that may not be

covered in the User's Guide or may only be covered relative to the part of their functionality that is being discussed. In many cases, you can access relevant parts of this document by pressing the help key.

Programmer's Reference Information

Additional documents provide details on Wind River and Eclipse APIs and libraries.

Help > Contents > Wind River Partners Documentation

viPlugin User Manual

The viPlugin allows you to add vi-type functionality to your Workbench editor as documented here. Additional information is available in the *viPlugin Help Center*, at **Help > Contents > viPlugin Help Center**.

Eclipse Workbench User Guide

This is the Eclipse documentation for Workbench and covers much of the functionality available in Workbench. For Wind River specific information, refer to the *Wind River Workbench User's Guide*.

Help > Contents > viPlugin Help Center

viPlugin Help Center User Manual v1.6

Additional information on using the viPlugin.

Help > Contents > C/C++ Development User Guide

C/C++ Development User Guide

The C/C++ Development Toolkit (CDT) is a collection of Eclipse-based features that provides the capability to create, edit, navigate, build, and debug projects that use C and/or C++ as a programming language.

Help > Contents > RSE User Guide

RSE User Guide

The Remote System Explorer (RSE) is a perspective and toolkit in Eclipse Workbench, that allows you to connect and work with a variety of remote systems.

PART II

Getting Started

2 Introducing Application Development 31

3 Creating and Debugging Platform Projects 49

4 Kernel Debugging (Kernel Mode) 87

5 Configuring Wind River Linux Platforms 115

2

Introducing Application Development

2.1 Introduction 31

2.2 Running Hello World in a Native-Mode Linux Environment 32

2.3 Creating a Cross-Debug Mode Project 37

2.4 Using the Editor 40

2.5 Configuring Project Build Properties 45

2.6 Building the Project and Correcting Simple Errors 46

2.1 Introduction

This chapter and the next three are tutorials on editing, building, and then running and debugging a cross-compiled application on a target system.

This chapter demonstrates default perspectives and views that you can use to compile and build a program. It explains how to use features of the editor such as name completion, bookmarks, and parameter hints, and how to find the beginnings and ends of code sections.

This first example requires that Workbench be installed on a Linux or Windows host computer. To learn more quickly, follow along on your computer.

2.2 Running Hello World in a Native-Mode Linux Environment

The following example uses the native development environment in which development tools such as **gcc** are on your host computer and in your **$PATH** setting. You do not need cross-development tools for native development.

If you are using a Windows host, go to *2.3 Creating a Cross-Debug Mode Project*, p.37.

Native-mode, *self-hosted* applications such as Hello World are created as Wind River Native Application Projects, Wind River User Defined Projects, and plain CDT projects. (See *6.4 Overview of Preconfigured Project Types*, p.146.)

2.2.1 Building the Hello World Application

If you have not already started Workbench, follow the steps in *1.2 Starting Workbench*, p.6.

To build **hello_world**:

1. In the Project Explorer near the upper left of the Workbench window (see Figure 1-1), right-click an empty area and select **New > Example**.
2. In the **Select a wizard** screen, choose **Native Sample Project** and click **Next**.
3. In the **Sample Project Template** screen, select **The Hello World Demonstration Program**, and click **Finish**.

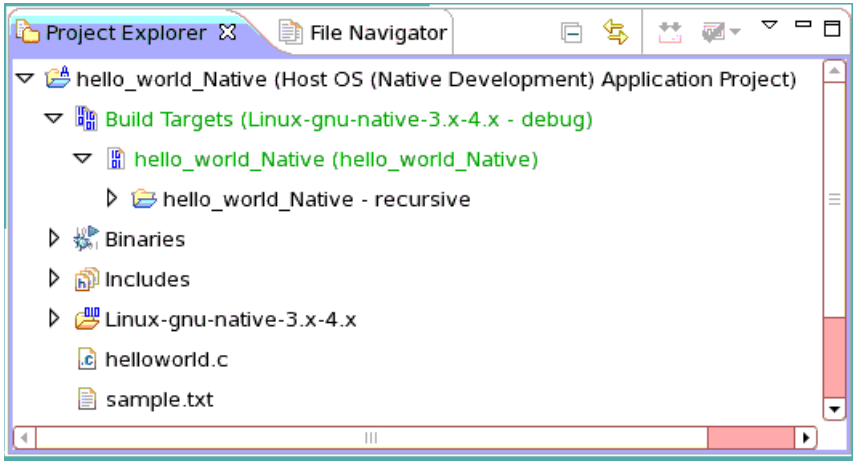
NOTE: Your particular project choices may differ.

Workbench creates the **hello_world** project. The sample project contains all the necessary files to build the program.

4. Right-click the **hello_world_Native** project folder and select **Build Project**.

   The executable is built. The Build Console at the bottom of the Workbench window displays build output.

5. Expand the *build target* (green container) in the Project Explorer to see the executable **hello_world_Native**. This is the *build output* or *build result*.



Your compiled application is now ready to be run on your target.

2.2.2 Debugging hello_world with the Wind River Debugger

To run and debug **hello_world**:

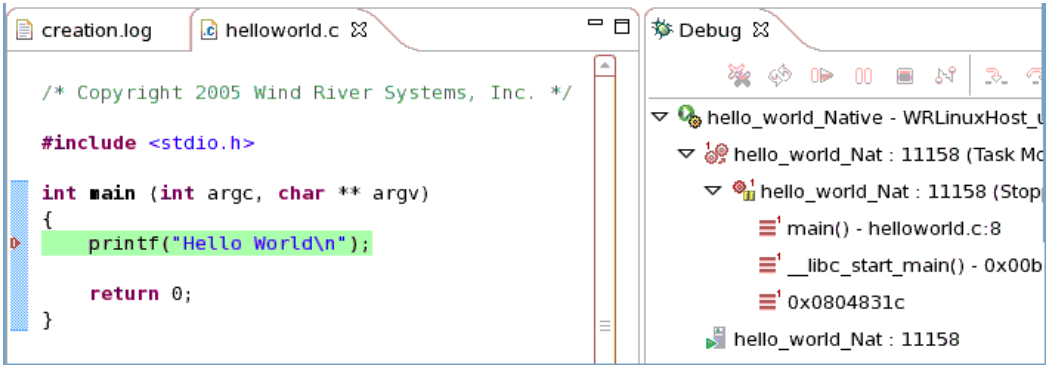
1. Create a connection to the target; for our example, this is the local Linux host:
   a. In the Remote Systems view (on the lower left), select **WRLinuxHost**_*username*.
   b. Click the green connection button in the Remote Systems toolbar to open a default output window. (The connection button looks like an "N" with round ends.)

   The **usermode-agent** window opens.
2. In the Remote Systems view, right-click **WRLinuxHost**_*username* and select **Debug > Debug Process on Target**.
3. In the Main tab of the Debug dialog, click the **Connect** button if **WRLinuxHost**_*username* is not connected. (It should be connected, and the button should be grayed out.)
4. In the Main tab, find the text box labeled **Exec Path on Target**. If there is no exec path, browse to the workspace location of the **hello_world** executable, select **hello_world**, and click **OK**.

(The location is *workspace*/**hello_world_Native/Linux-gnu-native-3.x/hello_world_Native /Debug/hello_world_Native**, or a similar path depending on your configuration.)

5. Click **Debug** in the lower right corner of the Debug dialog.

   The process executes until it comes to the **main( )** routine in the program. Workbench's Device Debug perspective shows the Debug view in the upper right. The editor displays the source file, and other views show typical debugging operations.



6. In the Debug view, note:
   - The process ID (PID) in the entry **hello_world_Nat**: *PID* **(Task Mode)**
   - The entry **hello_world_Nat**: *PID* **(Stopped - Breakpoint Hit)**
   - The breakpoint itself appears as **main( ) - helloworld.c:8**, meaning that execution has stopped at Line 8 in **helloworld.c**.
7. To display all the processes running on the target and the state of the **hello_world_Nat** process:
   a. Expand **Processes** in the Remote Systems view.
   b. Scroll down if necessary until you find the process ID of **hello_world_Nat**.
   c. Expand it.

   The process is shown as **[Stopped]** because it has hit a breakpoint.
8. Move to the Debug view and hover the mouse cursor over the buttons in the view's toolbar to see the debug functions that you can perform: Resume,

Terminate, Disconnect, Step (Into, Over, and Return), Toggle Assembly Stepping Mode, and Drop to Frame.

To see function keys assigned to common activities, click the **Run** icon in the main toolbar.

9. Click the **Step Over** button (or press **F6**). The program counter advances one line in the editor, and **Hello World** displays in the Console view. (You may have to click the Console tab in the lower middle window.)

   If you have added any variables to **helloworld.c**, and they have been assigned values, the lower-right Variables tab displays their current value(s).

10. Continue stepping through the program, or press **F8** or click the Resume button to complete it.

    When the program has completed, the Debug view displays its exit status: **<terminated, exit value: 0>hello_world_Nat:** *PID*.

To remove old information from the Debug view, click the Debug view's toolbar icon for **Remove All Terminated Launches**.

NOTE: You may also debug self-hosted applications from another Workbench on another host, in addition to using the local Workbench. For example, this lets you use a 64-bit host that you share with other, 32-bit hosts, so you can develop 64/32 bit applications when 64-bit hosts are rare.

To debug self-hosted applications from another host, start the usermode agent on the "self-hosted" host as if it were a target, then connect to it from another host. (See *3.5 Running Usermode Agent on Non-Emulated Targets*, p.65.)

2.2.3 Debugging Self-Hosted Applications with the CDT/GDB Debugger

You may use the GDB debugger built into Eclipse to debug native-mode, self-hosted applications such as Hello World.

To debug **hello_world** using the GDB debugger:

1. Right-click the **hello_world_Native** project folder.
2. Select **Debug As > Local C/C++ Application**.
3. Choose **gdb Debugger**.

The process executes until it comes to the **main( )** routine in the program. The Debug perspective shows the Debug view in the upper left. The editor displays the source file, and other views show typical debugging operations.

4. In the Debug view, note:
   - The process ID (PID) in the entry **hello_world_Nat**: *PID* **(Task Mode)**
   - The entry **hello_world_Nat**: *PID* **(Stopped - Step End)**
   - The breakpoint itself appears as **main( ) - helloworld.c:8**, meaning that execution has stopped at Line 8 in **helloworld.c**.
5. Move to the Debug view and hover the mouse cursor over the buttons in the view's toolbar to see the debug functions that you can perform: Restart, Resume, Terminate, Step Into, Step Over, and Instruction Stepping Mode.
6. Click the **Step Over** button (or press **F6**). The program counter advances one line in the editor, and **Hello World** displays in the Console view. (You may have to click the Console tab in the lower window.)

   If you have added any variables to **helloworld.c**, and they have been assigned values, the upper-right Variables tab displays their current value(s).
7. Step through the program, or press **F8** or the click **Resume** to complete it.

When the program has completed, the Debug view displays its exit status: **<terminated, exit value: 0>hello_world_Native [C/C++ Local Application]**.

To remove old information from the Debug view, click the Debug view's toolbar icon for **Remove All Terminated Launches**.

See *16.1 Introduction*, p.257, for breakpoints assigned to which debugger. Also see **http://download.eclipse.org/tools/cdt/docs/tutorials/debug_tutorial/cdt_w_debug.htm**.

2.3 Creating a Cross-Debug Mode Project

The Project Explorer lets you visually organize projects to reflect their inner dependencies, and therefore the order in which they are compiled and linked.

In this section, you create a project folder and import existing source files into it.

This project uses *cross-debug mode*, in which the output runs on an architecture different from your development architecture. The example supplied uses a PowerPC target and associated cross-development tools.

This section and the rest of this chapter use the **ball** sample program, written in C. This program implements a set of ASCII character balls bouncing in a two-dimensional grid. As the balls bounce, they collide with each other and with the walls.

The following chapter displays the Memory view showing the balls: see *3.9.5 Setting and Running to a Breakpoint*, p.78. The balls appear to move because you set a breakpoint with the property **continue-on-break** at the outer move loop, and you set a watch on the global **grid** array variable.

2.3.1 Checking and Setting Your Build Specs

Sysroots, which contain one or more *build specs*, provide the libraries, header files, and other files and directories that are essential to the infrastructure of your projects. The ones that came with your installation are automatically available in Workbench. These are in *installDir*/**wrlinux-2.0/sysroots**. You may put others in directories such as **~/myroots**, and specify the directory pathnames in Workbench.

To add sysroots, you must specify their directory:

1. Choose **Window > Preferences > Wind River > Wind River Linux**.
2. Click **Add** to the right of the Sysroots panel.
3. Browse to the location of your sysroots.
4. Click **OK**.

NOTE: Build specs are dynamically derived from the installed board templates or the extended templates from layers. It is not necessary to install additional sysroots for application development.

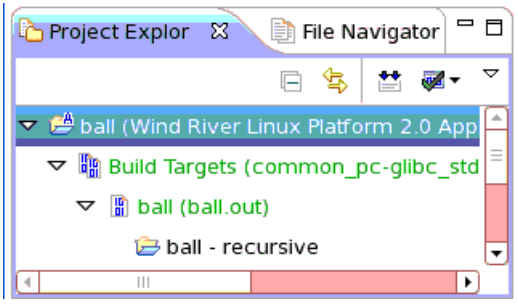
See *Installing Multi-lib Packages and Exporting Their Sysroots*, p.126, for multi-lib-specific information.

2.3.2 Creating a Wind River Linux Application Project

Create a new application project in your workspace as follows:

1. Right-click in the Project Explorer and and choose **New > Wind River Linux Application Project**.
2. Specify **ball** as the project name.
3. Follow *one of these two paths* to confirm you can view the active build spec and set it differently:
   - Click **Finish**, then right-click the project in the Project Explorer and select **Build Options > Set Active Build Spec**.

     Or:
   - Click **Next** and continue clicking **Next** until you come to the Build Specs window. In the **Active build spec** field, select **common_pc-glibc_std-i686**. Click **Finish**.
4. In the Project Explorer, click the arrow to the left of the **ball** folder to expand it to see what it includes**.**



2.3.3 Importing Existing Source Files

In the following procedure, you import source files into your new project.

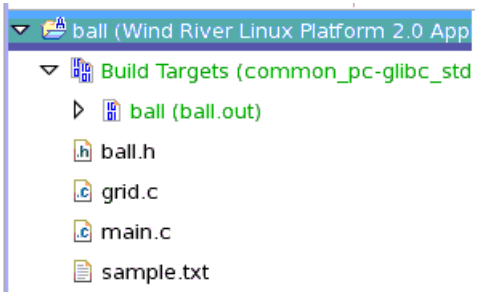
1. To import the existing source files of the ball program, right-click the ball folder you created in the previous procedure, then select **Import**.
2. In the **Import** dialog box, select **General > File System**, then click **Next**.
3. For the **From Directory** field, click **Browse**, then browse to *installDir*/**workbench-3.0/samples/ball** and click **OK**.

4. In the **File system** dialog box, select the check box next to the **ball** folder.

   The four files in that folder are now pre-selected in the right pane, and the **Into folder** field includes **ball**.

5. Click **Finish**.

The four new files are now included in the project in the Project Explorer:



2.3.4 Importing Build Specs

To add new build specs without impacting any existing build spec information:

1. Right-click the application project and select **Properties > Build Properties**.
2. In the **Build Support and Specs** tab, click **Import**.
3. In the Import wizard, select **Import build settings**, select **Default settings** in the list, and, below the list, select **Import specific build specs**. Click **Next**.
4. On the next page you will see the four default build specs provided in the Wind River Linux Platforms installation and any new sysroots you exported.
5. Select the desired new sysroot and click **Finish**.

You have now imported a single new sysroot into the existing application project.

2.4 Using the Editor

In this section, you exercise various aspects of the built-in editor before moving on to the next section to build the project.

(If you want to skip this introduction to the editor, continue with *2.6.3 Rebuilding the Project*, p.47.)

The editor can emulate the **vi** or **emacs** editors. To set your preference, click the appropriate icon in the top toolbar. Refer to additional editor preferences in **Window > Preferences > General > Editor**, and to additional online information at **http://help.eclipse.org**.

2.4.1 Changing File Preferences

To open a source file and change its properties (for example, its font properties):

1. Expand the **ball** folder and double-click **main.c**.

   The **main.c** file displays in the editor, using a fixed font and preference-based color syntax highlighting.

2. Select **Window > Preferences > General > Appearance > Colors and Fonts**.
3. Expand the **Basic** folder in the **Colors and Fonts** pane and select **Text Font**.
4. Click **Change** and select:
   - Family: Courier 10 Pitch
   - Style: Regular
   - Size: 10
5. Click **OK**.
6. Move the **Preferences** dialog box out of the way to uncover the **main.c** editor window if necessary. Click **Apply** to see the changes without closing the dialog box.
7. Explore and experiment with other preferences. Click **Restore Defaults** if you prefer the default settings.
8. Click **OK**.

2.4.2 Navigating in the Source

You can find your way in source files using:

- The Outline view, which lists elements and symbols in the file in focus in the editor
- **Find** utilities to locate symbols, elements, and strings

Using the Outline View

In the Outline view, click any item in the list to focus the editor on its declaration. Hover over the buttons in the toolbar to see how to filter the list of names in the view. You can sort them, hide fields and static members, and hide non-public members. (Note that the elements are sorted within types.)

The Outline view is limited to the names declared in the file that is open in the editor. To find the declaration of any symbol appearing in the editor, right-click the symbol, in the Editor view, then click **Open Declaration**.

For example:

1. In the Outline view, click **main(void): int**.
2. In the editor, click the call to **gridInit( )** to highlight that line.
3. Right-click **gridInit( )** and select **Declarations**. Specify the scope for your search: workspace, project, or working set.

   The Search tab displays the ball node, which you can open onto a list of the project files in which **gridInit( )** has been declared.

Finding Elements

To open a more advanced symbol or element search dialog box:

1. Select **Navigate > Open Element**.

   The **Open Element** dialog appears.
2. Enter **grid*Ball**.

   As you enter a **Pattern** for the symbol, including wild cards, Workbench lists all matching elements. All those that match **grid*Ball** are displayed.
3. Highlight one and press **Enter** to see its declaration.

Finding Strings

To find and optionally replace any string in the active Editor view, use these functions:

- **Edit > Find/Replace** (or use **CTRL+F**)
- **CTRL+K** to **Find Next**
- **CTRL+SHIFT+K** to **Find Previous**

See the **Edit** menu for other choices.

The **Search** menu provides a **grep**-like search for strings in any file in the workspace or in any location.

2.4.3 Using Code Completion to Suggest Elements

Code completion suggests methods, properties, and events as you enter code.

To use code completion, right-click in the editor and select **Source > Content Assist** or use **CTRL+SPACE** as a keyboard shortcut. A pop-up list displays valid choices based on the letters you have typed so far.

For example, with the **main.c** file in the editor:

1. Position your cursor inside **main( )** to the right of the first "**{**" character and press **ENTER**.
2. Type the first two characters of **grid** and then invoke code completion by pressing **CTRL+SPACE**.

   A dialog box appears with suggestions.

   As you continue to type, your choices narrow.
3. Select **gridAddBall(BALL*, pBall,point point void)** from among the choices, then press **ENTER** to add the routine to the cursor location in the editor.

2.4.4 Getting Parameter Hints for Routine Data Types

Parameter hints describe the data types that a routine accepts.

To see the data type descriptions for a routine's parameters:

1. Type a routine name followed by an open parenthesis, for example, **ballNew(**.

When you type the open parenthesis, the editor supplies the closing one, and places your cursor between the two.

2. **Press CTRL+SHIFT+SPACE**.

   The parameter hints appear, reminding you of the parameter types defined by the routine. You can copy and paste the hints if you wish.

2.4.5 Finding Beginning and End of a Code Section

Bracket matching lets you find the end (or beginning) of sections of code such as functions, comments, and so on.

Bracket matching operates on the following characters: **( )** (parentheses); **[ ]** (brackets); **{ }** (braces); **" "** (quotes); **/* */** (comments); and **< >** *(C/C++ only)*.

To use bracket matching:

1. Position the cursor after one of the bracket matching characters, either the opening or closing character.

   A rectangle of very thin lines encloses the matching character. If your cursor is after an opening character, the rectangle appears after the corresponding closing one. If after a closing character, it appears after the corresponding opening one.

2. Press **CTRL+SHIFT+P**.

   The cursor moves to just after the matching character.

3. Press **CTRL+SHIFT+P** a second time.

   The cursor returns to the original position.

NOTE: Before you proceed with the tutorial, undo any changes you might have made to **main.c** by selecting **File > Revert**.

2.4.6 Using Bookmarks

Adding a bookmark to a source file is similar to placing a bookmark in a book: it allows you to find an item you are interested in at a later time by looking in the Bookmarks view. Open the Bookmarks view by selecting **Window > Show View > Bookmarks**.

You can bookmark a file or a particular line of code within it.

The following procedures introduce an error in the source code and bookmark it. Later procedures explain how to recognize that the build does not complete because of the error; how to find the error; and how to rebuild the project.

Introducing an Error for this Tutorial

To introduce an error:

1. Find **main( )** in the **main.c** file in the editor.
2. Move down a few lines to find the call to **gridInit( )**.
3. Delete the semicolon after the call to **gridInit( )**.

Creating the Bookmark to the Error

To create the bookmark to it:

1. Right-click the left gutter of the editor next to that line, then select **Add Bookmark**.

   (The *gutter* is the shaded strip at the left edge of the Editor view. It is likely in blue, and it may be very thin.)
2. In the **Add Bookmark** dialog box, enter a bookmark name (or accept the default **gridInit( )**, and click **OK**.

   A small bookmark icon appears in the gutter.
3. To save the file with the error, click the **Save** button on the main toolbar.
4. Close all open files by clicking the **X** on the tab of their Editor view.

Locating and Viewing the Bookmark

To locate and view bookmarks:

1. To open the Bookmarks tab, select **Window > Show View > Other > General > Bookmarks**.

   The Bookmarks view shows all bookmarks in the project. Because there is only one bookmark in this project, only one bookmark appears in the list.
2. Double-click the entry. If you accepted the default name, it is **gridInit( )**.

The **main.c** file opens in the editor with the bookmark line highlighted.

3. Use the **X** in the **main.c** tab to close the file without making any changes.

   (That is, leave the error so you can follow the tutorial steps below, which include correcting errors in builds).

2.5 Configuring Project Build Properties

Every project has a set of build properties, which you can use to change target specifications, relocate build output, see and modify the actual build command line, and more.

To configure project properties:

1. Right-click the project, for example, **ball (Wind River Linux Platform 2.0 Application)**.
2. Choose **Properties**.
3. Click **Build Properties** in the left of the **Build Properties** dialog.

   If there is no **Build Properties** choice listed between the entries Resource and Builders, you have not specified the build specs for your project. See *2.3.1 Checking and Setting Your Build Specs*, p.37.
4. Set the active build spec, other build support items such as debug mode and managed build, and any other appropriate settings under the various other tabs such as **Build Support and Specs**, **Build Tools**, **Build Macros**, **Build Paths**, and **Libraries**.

   For the current project example, you should not need to specify any changes, but they are available for reference.

   Macro values change according to the active build spec, and you would not typically modify them. LAYER is the directory from which the sysroot was defined, for example *installDir*/**wrlinux-2.0**.

   Use the Libraries tab to specify library search paths and filenames, such as **–L** *path*; and, for libraries, **–l** *lib*.
5. Click **OK** to save any modified build properties.

The build target after **Build Targets** in the Project Explorer is your build spec name.

2.6 Building the Project and Correcting Simple Errors

If you completed *2.3 Creating a Cross-Debug Mode Project*, p.37, the ball project folder includes the files needed to build and debug the program. In *2.2 Running Hello World in a Native-Mode Linux Environment*, p.32, you chose to run on the local host because the target connection was the localhost connection.

In this example, you build the program with cross-development tools that generate a build target for your target architecture. The example uses a Power PC target, but you can substitute your target architecture if it is different. Depending on your Wind River Linux installation, the *installDir*/**gnu** directory contains appropriate tools for your licensed architecture, including the C language cross-compiler.

2.6.1 Building a Project with Introduced Errors

In *2.4.6 Using Bookmarks*, p.43, you introduced an error in the ball program source code that you will fix in the following procedure, which demonstrates how to find and fix build errors using Workbench.

To fix build errors:

1. In the Project Explorer, right-click the ball folder and select **Build Project**.
2. Click **New Build Target** and click **Finish** in the resulting Build Target dialog (thereby accepting the default names).
3. Click **Continue** if a prompt appears concerning the include search path

NOTE: If necessary, add **/usr/include** to the include search path.

Build output displays in the Console tab, and entries also appear in the Problems tab. Click these tabs to move back and forth between their contents (or rearrange your window to view them both simultaneously).

Because you created an error in the **main.c** file, errors are encountered during the build process. Notice that Workbench enters a task in the Problems view showing the type of error, the file the error is in, the location of the file, and the location of the error. It also shows warnings that occur in other locations in the code because of the error.

4. Double-click the error line that begins **error expected ';'** in the Problems view.

   The editor focuses on the erroneous line, with an identical white **X** in a red circle to mark the position of the error in the file.

5. Re-enter the semicolon (;) character you deleted in earlier steps.

   This was at Line 62 in **main.c**, after **gridInit( )**.

6. Right-click the bookmark icon in the gutter and select **Remove Bookmark**.
7. Save and close the file.

2.6.2 Displaying a File's History

Workbench tracks all changes that are made to any files in the project.

At this point, several changes have been made to the **main.c** file.

To display the change history on the **main.c** file:

1. Right-click the file in the project tree and select **Compare With > Local History**.

   The History tab opens. (It may be labeled just "H.") The upper area of the History view displays a list of the dates and times when the file was changed.

2. Double-click one of the list entries.

   A Comparison Viewer dialog opens, displaying the file on the left as it was at the date and time of the list entry that you double clicked. To the right of the file display is the file as it was in the immediately preceding version.

3. Move your cursor over the Compare Viewer's toolbar icons to read the functions provided, such as copying changes from the left to the right file, moving to the next difference or the next change, and so on.
4. Close the Compare Viewer.

   If you see a Save Resource error notice that **main.c** has been modified but is still open, click **Yes** to save the changes you made.

2.6.3 Rebuilding the Project

To rebuild the ball project:

1. Right-click the ball folder at the top of the project tree.
2. Select **Rebuild Project**.

Now the project builds without errors.

3

Creating and Debugging Platform Projects

3.1 Introduction 50

3.2 Creating a Wind River Linux Platform Project 54

3.3 Building the Target's File System and Kernel 57

3.4 Configuring Target Connections 61

3.5 Running Usermode Agent on Non-Emulated Targets 65

3.6 Testing Target Connections 67

3.7 Using the Remote System Explorer 67

3.8 Using the One-Step Launch Configuration 69

3.9 Debugging an Application on the Target 72

3.10 Customizing Builds with Layers and Templates 81

3.11 Working with External Locations 84

3.1 Introduction

This chapter uses tutorial examples to describe how to create a new Wind River Linux platform project, and how to deploy and debug it on a target. It explains how to configure and test target connections, and how to create projects at external locations.

With the exception noted below, you cannot debug an application or a kernel until you have configured a platform, and have booted the platform with the kernel and file system that you have configured. Then, you can start debugging an application or the kernel itself.

NOTE: The only exception to this is native-mode (self-hosted) debugging, where you do not have to configure a target platform because the development host *is* the target platform. (You can only do this on a common PC running Linux.)

This introductory section contains the following subsections, which contain cross references to other parts of this chapter:

- *3.1.1 Checklist for Creating a Platform Project to Debug*, p.51
- *3.1.2 Exploring Targets and Making Easier Connections*, p.53
- *3.1.3 Debugging Applications*, p.53
- *3.1.4 Creating External Projects and Using Layers and Templates*, p.54

3.1.1 Checklist for Creating a Platform Project to Debug

This chapter covers many topics that include a large range of complex features and ideas. You can always come back to the following checklist at various points to reconnect to the main thread that runs through these features.

On the other hand, you may be able to follow the checklist by itself.

NOTE: Some of the topics below have been explained in the previous chapter, or, as with Steps 3 to 4, *within this checklist itself*.

Essentially, you want to take the following steps to create and debug a platform project:

1. Create a Wind River *platform project* and do a build target on its FS node to create a file system for the platform. See *3.2 Creating a Wind River Linux Platform Project*, p.54.

   These steps create a *ProjectName*_**prj** directory containing the file system for the remote system (target), and a default kernel. The file system and kernel are on the host and are not, at this point, on the target system or accessible by it.

2. Create a Wind River Linux *application project* from among the available pre-coded sample applications such as Hello World, Penguin, and Multi-Thread. See *2.3.2 Creating a Wind River Linux Application Project*, p.38, in the previous chapter, for an example.

NOTE: If you want to follow this chapter as a tutorial, build the ball application for the ARM Versatile AB-926EJS.

3. Configure the build paths information for the *application* by using the application's **Properties > Build Properties > Build Paths** to specify a redirection root directory, to contain the application build results.

   For example, you can specify a redirection root directory of *ProjectName*_**prj/export/dist**, which results in the creation at build time of *Application*/*BuildSpec*/*Application*/Debug/*Application*.out within it.

4. Build the application project.

   Its output is placed in the redirection root directory, such as configured in the above step. For example, **mthread** appears in the **export/dist** directory.

5. Define a connection to the target:

   a. Click the **Define a connection** button in the Remote Systems view toolbar.

b. Choose the appropriate type of connection to create:

- For *real targets*: click **Wind River Linux User Mode Target Server** and specify the hostname of the target, the port number to use for communicating with it, and the location of the root file system. See *3.4 Configuring Target Connections*, p.61.

  If you specify **localhost** and Port **4444**, and follow the directions for real targets, Workbench creates a QEMU connection for you, but this requires more manual intervention. Also, the port number depends on the instance of QEMU that is running. For an instance of 0, the port is 4444; for an instance of 1, it is 4544: for each increase in the instance number, the port number increases by 100.

- For *QEMU-emulated targets* (for this tutorial): click **Wind River QEMU Linux User Mode Target Server**. Workbench simplifies many aspects of creating and using QEMU targets if you follow this approach.

6. Specify, or verify, for the connection, *object path mappings* that name the target path (always **/**) and host path, such as *workspace*/*ProjectName*_**prj/export/dist**.

   This mapping specifies which host directory is to be NFS mounted on the target's root directory (or, in the case of QEMU, which directory is to be mapped to the emulated target's root directory).

NOTE: Make sure that you have specified the root directory, for example, *workspace*/*PlatformName*_**prj/export/dist**, in the application's root directory build property. If you do not see your *application* subdirectory in the **dist** directory, rebuild the application project as in Step 4.

7. For non-QEMU targets:

   a. Start your target: power up the board and wait until you see the kernel (target) shell present a window on the host via an Ethernet, serial, or USB connection, then configure for the kernel and root file system, and re-boot.

   b. Run **/usr/bin/usermode-agent -daemon** in the target window (see Step 3 in *3.5 Running Usermode Agent on Non-Emulated Targets*, p.65).

      The usermode agent waits for connections and handles the Wind River debugging (WDB Agent) communication between Workbench and the target.

   c. Connect the target connection to the usermode agent running on the target, by selecting the connection and clicking the Connect icon in the Remote Systems toolbar. (See *3.6 Testing Target Connections*, p.67.)

8. Run or debug the application on the target by right-clicking it, choosing **Run Process on Target** or **Debug Process on Target**, and then clicking **Run** or **Debug** in the lower-right of the Create, Manage, and Run Configurations screen. (See *3.9 Debugging an Application on the Target*, p.72.)

3.1.2 Exploring Targets and Making Easier Connections

Once you have the connection to the target, you can use what you have created or configured at this point to:

1. Use the Remote System Explorer to browse a remote target's file system; browse your host's target system; copy files from directory to directory within the target; and issue remote shell commands to your target. (See *3.7 Using the Remote System Explorer*, p.67.)
2. Modify and use the default launch configuration as a one-step process to connect to your target and launch your application to run or debug it. (See *3.8 Using the One-Step Launch Configuration*, p.69.)

3.1.3 Debugging Applications

Now that you can connect to the target and launch your application on it so you can debug it, you can use the debugging section of this chapter to:

- Recognize the parts of the Workbench debugging environment (*3.9.1 The Workbench Debugging Environment*, p.72)
- Run a program on a target until it reaches the first breakpoint (*3.9.2 Running on the Target up to a Breakpoint*, p.73)
- Use the device debug perspective (*3.9.3 Using the Device Debug Perspective*, p.74)
- Step through code and examine memory (*3.9.4 Stepping Through Code*, p.76)
- Set and modify breakpoints (*3.9.5 Setting and Running to a Breakpoint*, p.78, and *3.9.6 Modifying the Breakpoint to Execute Continuously*, p.80)

3.1.4 Creating External Projects and Using Layers and Templates

In addition to creating, running, and debugging applications, you can:

1. Create projects and build usermode-agent at external locations, which are external to your workspace directory (see *3.11.1 Creating User-Defined Projects at External Locations*, p.84, and *3.11.2 Building usermode-agent at External Locations*, p.85)
2. Use layers and templates to customize builds for purposes such as: adding packages; creating new BSPs; ensuring that custom builds can be easily repeated; and completely separating custom code from the installed Wind River development environment (see *3.10.1 Adding Templates to a Project*, p.82, and *3.10.2 Adding Layers to a Project*, p.83)

3.2 Creating a Wind River Linux Platform Project

To create a new platform project:

1. Right-click in a blank space in the Project Explorer and select **New > Wind River Linux Platform Project**.

   When you use Workbench to create a new Wind River Linux Platform project, the choices you see in the project wizard are based on your installation.

2. Enter a name for the project and click **Next**.

   To follow the tutorial and see the same results, use **ball** for the project name. You may have to delete the ball project created in the previous chapter.

3. Depending on your installation, you will have different options for selecting a build spec, a root file system, and a kernel. Select:

   - A board from the **board** menu.

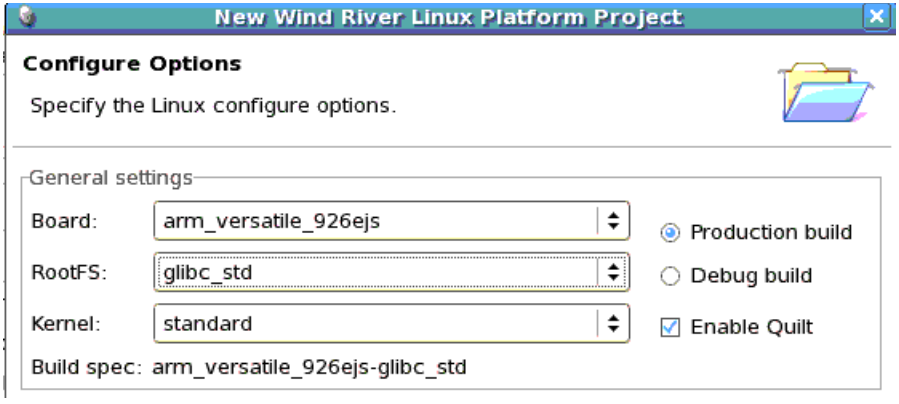
NOTE: Check the Release Notes and the Online Support site to see which boards are emulated by QEMU. Currently, these include the Common PC and ARM Versatile AB-926EJS boards, among others.

   - A file system size from the **RootFS** menu. Possible options are **glibc_cgl** for carrier grade Linux, **glibc_std** for standard Linux, **glibc_small** for a

Busybox-based file system, and **uclibc_small** for one based on **uClibc**. (To follow the tutorial example, choose **glibc_std**.)

- A kernel. Possible options are **cgl**, **ecgl**, **standard**, **small**, **preempt_rt** (the Open Source soft realtime kernel), and **rtcore** (Wind River's guaranteed real time kernel), depending on the choice of RootFS.

The build spec associated with that board, kernel, and root file system is displayed.



4. *Optional and not needed here*: If you want to add or modify arguments to the **configure** command, select **Advanced > >**. You can then click **Add** for the **Option/Value** window and enter options directly by selecting them from the list provided, and editing them as appropriate. Substitute actual values for **DIR**, **BUILD**, and so on, and choose between options specified in square brackets and separated by lines. For example, if you select the following option:

   ```
   --with-toolchain-version=[STRING|current]
   ```

   you would substitute a value for **STRING** or use "current" as in:

   ```
   --with-toolchain-version=current
   ```

   If you select an argument from the list that contains a **DIR** value to be replaced, the **Browse** button becomes enabled and you can browse to choose the appropriate directory. Click **Add** to add each additional option.

   When you have finished adding options, click **OK**.

   For now, ignore the Configure Options dialog box. Proceed with the next step.

5. *Optional and not needed here*: If you are in the **Advanced > >** view of the **Configure Options** dialog, you can click **Add** for **Layers** or **Templates**. See *3.10 Customizing Builds with Layers and Templates*, p.81.

   For now, ignore layers and templates. Proceed with the next step.

6. Click **Finish** to ignore the source code indexer and its associated performance penalty.

   The **configure** command is executed. Its output appears in the Build Console and is stored with additional information in the project's **creation.log** file.

   A new platform project creates two subdirectories in your workspace—one with the project name (*project*) and one with the project name plus a **_prj** extension (*project*_**prj**).

NOTE: If you are importing an existing platform project, its location is used, and a *project*_**prj** folder will not be created.

   At this point, your *project*_**prj/export** directory contains three empty subdirectories: **dist**, **packages**, and **RPMS**.

You can now build the kernel, file system, and other targets, as described next. Before building your target, you could also:

- Reconfigure kernel parameters as described in *5.2 Configuring Wind River Linux Platform Kernels*, p.116.
- Add or remove RPMs from the file system as described in *5.5 Working with Packages and Target File Systems*, p.121.

For the purposes of this tutorial, just continue with the next section.

3.3 Building the Target's File System and Kernel

Your platform project needs to provide a file system and kernel for the target.

3.3.1 Building the File System

To build the file system:

1. Right-click **fs** within the Project Explorer, and choose **Build Target**.
2. Look in the Build Console for reports on the build progress. The build might take many minutes.
3. Look toward the very end of the **creation.log** window in the center of Workbench for a summary of what you have built.

To see what you have built, in more detail:

1. Open a terminal window.
2. Change directory (**cd**) to *workspace*/*ProjectName*_**prj**.
3. List the **export** and **export/dist** directories.
   - The **export** directory contains:

     The compressed runtime file system as a **tar.bz2** file named according to the board, rootfs, and kernel you chose, for example: **arm_versatile_926ejs-glibc_std_std-dist.tar.bz2**.

     A link to the default kernel, for example, **arm-versatile_926ejs-default_kernel_image_WR2.0au_std**.
   - The **export/dist** directory contains the file system for your target. Later, it will contain the build output from the application(s) that you create.

3.3.2 Building the Kernel

The project you have created contains a default, pre-built kernel, so *you do not actually have to build a kernel at this time*. However, you may do so.

To build a kernel for your project:

1. Right-click **kernel_build** within the project's folder in Project Explorer. Choose **Build Target**.

2. Look in the Build Console for reports on the build progress, such as patch related steps, configuration options, many object (**.o**) files, and so on, until you get to "Creating rpm: linux," followed by a few dozen lines in red.

   The build time depends on the size of the kernel source tree. For the arm_versatile_926ejs board and std kernel, this may be 20 minutes or more.

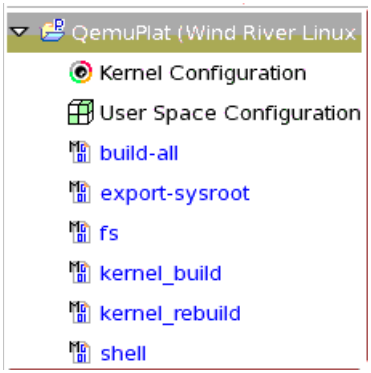
To find what you have built:

1. Open a terminal window.
2. Change directory (**cd**) to *workspace*/*ProjectName*_**prj**.
3. List the **export** and **export/dist** directories, for example:
   - The **export** directory contains a link named, for example, **arm_versatile_926ejs-zImage-WR2.0au_std**, to the actual kernel in *project*_**prj/build/linux-2.6.21-std/arch/arm/boot/zImage**.
   - The **export** directory also contains the kernel symbol file, for example, **arm_versatile_926ejs-vmlinux-symbols-WR2.0au_std**, a link to the actual file *project*_**prj/build/linux-2.6.21-std/vmlinux**.

   (The export directory actually contains more files than those just described.)

3.3.3 Understanding What Else You Can Build

This section provides an overview of what appears in a platform project's folder, part of which is shown below.



The *project folder* includes the following nodes: kernel configuration, user space configuration, build, and project files:

- **Kernel Configuration** node—Right-click and choose **Edit Linux Kernel Configuration** to view and modify the kernel's **.config** file (see *5.2 Configuring Wind River Linux Platform Kernels*, p.116).
- **User Space Configuration** node—Right-click and choose **Edit User Space Configuration**, then click **Packages** or **Filesystem** (the lower-left tabs in the central **Packages** window) to add or remove file system RPMs and to see the **export/dist** target file system. (see *5.5 Working with Packages and Target File Systems*, p.121).
- Blue build nodes—See the following table for what you can build or accomplish when you right-click one of the blue build nodes and select an option. Note that some build nodes are available only for projects for which you have selected the configure option for flash files.

Table 3-1 **Build Nodes and Right-Click Actions**

Build Node Options	To Build . . . Or to . . .
build_all	All tools and the file system.
cramfs	A CRAMFS file system. Only present for Flash.
delete	Clean the project file *before removing a project* by right-clicking the project folder and selecting **Delete**.
deploy	Deploy the project as described in *5.7 Automating Target Deployment*, p.138.
export-layer	Extract the changes to a project into a shareable layer directory in the export directory, with the timestamp in its name, and added to source control. See *3.10 Customizing Builds with Layers and Templates*, p.81.
export-sysroot	Create a sysroots directory in the export/sysroot directory, which can be used for providing build specs. See *2.3.1 Checking and Setting Your Build Specs*, p.37.
fs	A file system for your target.
import-package	Bring up a GUI applet to help you add external packages to a project.

Table 3-1 **Build Nodes and Right-Click Actions**

Build Node Options	To Build . . . Or to . . .
jffs2	A JFFS2 file system. Only present for Flash.
kernel-build	The kernel.
kernel-clean	Perform a clean on the kernel directory, equivalent to **make -C build linux.clean**.
kernel-config	Configure the kernel using **config**.
kennel-menuconfig	Configure the kernel using **make menuconfig**.
kernel-rebuild	Rebuild the kernel.
kernel-xconfig	Configure the kernel using **make xconfig**.
ramfs	A RAMFS file system. Only present for Flash.

NOTE: The Build Console displays the full **make** command and output when building the targets.

- *Target* node—A link to the template directory for the respective board. See *Wind River Linux Platforms User's Guide* for more information.
- Additional files—Those in the following table are also included.

Table 3-2 **Additional Files in the Project Folder View**

File Name	Contents
creation.log	Detailed **configure** command output.
Makefile	The Wind River Workbench-generated **Makefile** for the project..
Makefile.wr	Make rules for the build targets, used in place of **.wrmakefile**.

- Targets you specify: **Project > Properties > Build Properties > Build Targets**.

The *project*_**prj** *directory* in the host file system holds the actual project contents such as the **dist** and **export** folders. This directory is identical to a build directory created by the command line tools and can exist outside of the workspace folder (as, for example, when an existing project is imported into Workbench), in which case, the platform project has an Eclipse link to the actual location.

3.4 Configuring Target Connections

This section explains how to create and configure connections to:

- The QEMU (Quick Emulator) target emulator, using a QEMU connection
- An actual hardware target

3.4.1 Creating and Configuring QEMU Connections

If you are connecting to an actual, non-emulated hardware target, skip to *3.4.2 Configuring Hardware Target Connections*, p.63.

QEMU, an open source machine emulator written by Fabrice Bellard, allows you to configure the kernel and root file system for emulated boards for which Wind River has created board support packages (BSPs) such as the Common PC and ARM Versatile AB-926EJS.

QEMU loads the kernel found in the project build directory's **export** directory and NFS mounts the file system within the **export/dist** directory. Unlike configuring hardware targets, for which you can place the kernel and file system in any location on your host system to be NFS mounted, QEMU requires specific locations.

To create and configure a QEMU connection for an emulated target:

1. Create a QEMU target server connection:
   a. Right-click in the Remote Systems view and select **New > Connection**.

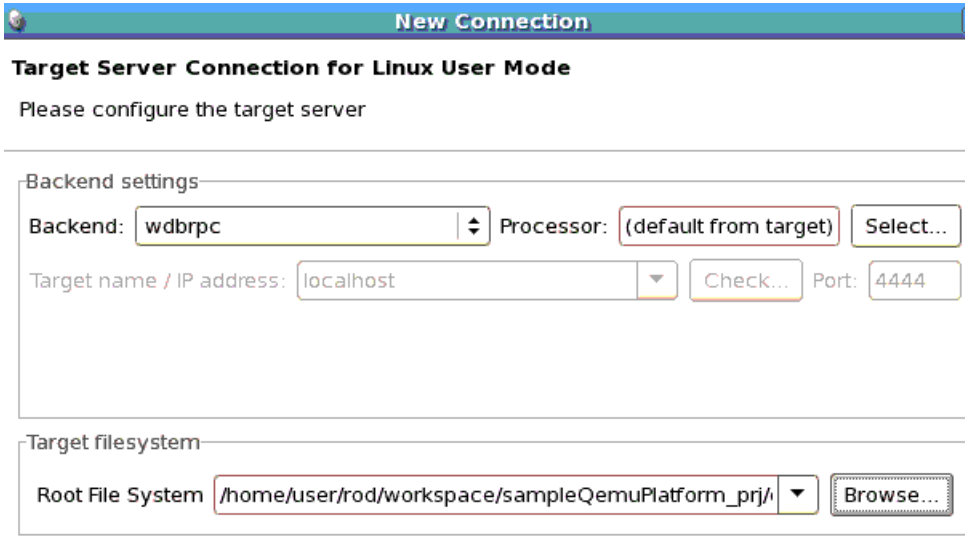
      The Select Remote System Type dialog appears.

   b. Select **Wind River Linux > Wind River QEMU Linux User Mode Target Server Connection** then press **Next**.

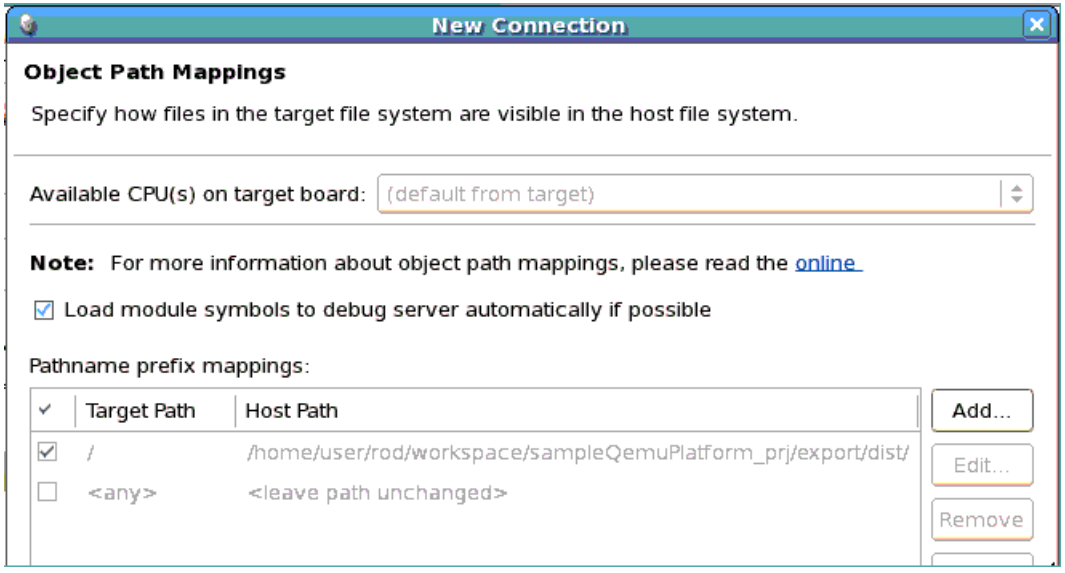
      The QEMU Simulator Configuration screen appears.
2. Specify the project's root directory or browse to *workspace*/*projectName*_**prj**. Click **Next**.

   (For example, enter **/home/***user***/workspace/ballPlatform_prj**.)

   The Target Server Connection for Linux User Mode dialog appears.
3. Specify the root file system: *workspace*/*projectName*_**prj/export/dist** (this is the file system to be exported to QEMU). Click **Next**.



The Object Path Mappings screen appears:


New Connection
Object Path Mappings
Specify how files in the target file system are visible in the host file system.
Available CPU(s) on target board:
(default from target)
Note: For more information about object path mappings, please read the online
Load module symbols to debug server automatically if possible
Pathname prefix mappings:
Target Path
Host Path
/
/home/user/rod/workspace/sampleQemuPlatform_prj/export/dist/
<any>
<leave path unchanged>
Add...
Edit...
Remove

The object path mappings determine which part of the host file system gets NFS mounted or mounted by QEMU on the target.

4. If the target path is not **/**, or the host path is not your *platform*_**prj/export/dist** directory, use the **Add** button to specify them, and select the single box for the pair you want to use.
5. Click **Finish**.

   The **Virtual-WRLinux0** terminal window opens. It reports the steps in the process of making the connection. In the Remote Systems view, a connection appears: **WRQEMULinuxUser**_*Project*_**prj**. This is the target connection you will use in *3.9 Debugging an Application on the Target*, p.72.
6. You may have to log in as **root** with the password **root**, unless you are using a small rootFS, in which case you are automatically logged in as root.
7. Continue with *3.6 Testing Target Connections*, p.67.

3.4.2 Configuring Hardware Target Connections

When you use an actual external target over the network, you must configure the target to locate its kernel and file system.

Typically, a boot loader on the target will download a kernel from your host, and the target will NFS mount a root file system from your host.

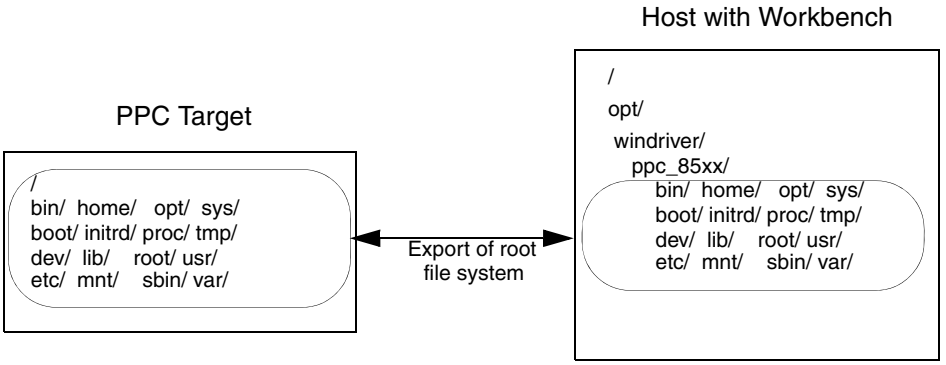
This section assumes the target board has already been configured but gives an overview of how your kernel and file system might be configured for access by your target board. Refer to the *Wind River Linux Platforms User's Guide*, the installation-specific **README** files. Also, see your target's boot documentation for details on how to supply boot parameters.

Configuring NFS to Export Root File Systems to the Target

Consider an example where **/opt/windriver/ppc_85xx** is exported to a target named **ppc_target**, from where it is mounted as the root file system (**/**).

Figure 3-1 illustrates an exported root file system. (Note that changes in either location are directly reflected in the other view.)

Figure 3-1 **Exported Root File System Example**



Alternatively, if you are using a target that has its own file system, you can NFS mount that file system on your host, for example:

```
# mount -t nfs targetbox:/ /opt/windriver/ppc_85xx
```

Configuring TFTP to Provide a Kernel

Many boards use TFTP to download a kernel from a network server. If this is how you are providing the kernel, copy the kernel from the **export** directory in your project build directory to the TFTP directory, typically **/tftpboot**.

Creating an Actual Target Connection

To create a connection to your target:

1. Click on the **Define a connection to remote system** icon in the Remote Systems toolbar.
2. Select **Wind River Linux User Mode Target Server Connection**.

   Click **Next**.
3. Enter the IP address or hostname of the target in the **Target name / IP address** field. Click **Finish**.

   A new target connection entry should appear in your Remote Systems view.

4. Right-click the target connection and choose **Properties**.
5. In the Target Server Options tab, specify the target name and IP address, and also the root file system, that is, the directory that should receive the build output.

   For example, If you are configured as shown in Figure 3-1, you could enter **/opt/windriver/ppc85xx** for the root file system.
6. In the Object Path Mappings tab, If the target path is not /, or the host path is not your root file system directory, use the **Add** button to specify them, and select the single box for the pair you want to use.
7. Click **OK**.

3.5 Running Usermode Agent on Non-Emulated Targets

When you created the localhost connection to run Hello World in native-debug mode (see *2.2 Running Hello World in a Native-Mode Linux Environment*, p.32), Workbench started a program called **usermode-agent** on your host. This program is used in Workbench communications between target and host.

In cross-debug mode, the agent must run on your target before Workbench can connect to it.

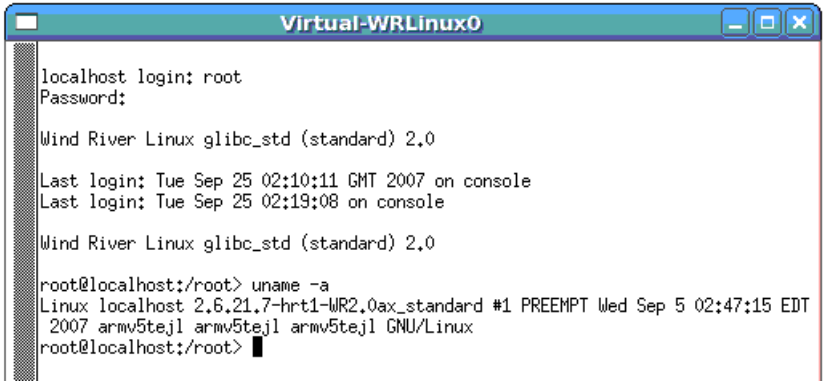
NOTE: If you have configured a small root file system project, **usermode-agent** is not automatically supplied for it. You need to add the template **feature/usermode-agent**. See *3.10.1 Adding Templates to a Project*, p.82. Workbench adds the debug template automatically when you select a small file system.

To run the agent on your non-emulated target:

1. Boot the target.
2. Open a terminal window on it via an Ethernet, serial, or USB connection.

   You can run the usermode agent as a user or as root, but the processes you launch from Workbench will run as the user that started the agent on the target.

Just to verify that you are on the appropriate target, use **uname -a** to display diagnostic information, such as the kernel identification:


Virtual-WRLinux0
localhost login: root
Password:
Wind River Linux glibc_std (standard) 2.0
Last login: Tue Sep 25 02:10:11 GMT 2007 on console
Last login: Tue Sep 25 02:19:08 on console
Wind River Linux glibc_std (standard) 2.0
root@localhost:/root> uname -a
Linux localhost 2.6.21.7-hrt1-WR2.0ax_standard #1 PREEMPT Wed Sep 5 02:47:15 EDT
2007 armv5tejl armv5tejl armv5tejl GNU/Linux
root@localhost:/root>

3. Execute the usermode-agent program:

   ```
   $ /usr/bin/usermode-agent -daemon
   ```

   (If your target is using the **uClibc** library, add the **-no-threads** argument.)

   You should see the terminal window display a WDB Agent 3.0 Ready message.

   ```
   root@localhost:/root> /usr/bin/usermode-agent -daemon


          WDB AGENT 3.0 DAEMON READY
   (c) Copyright Wind River Systems Inc. 2007
   All rights reserved.

   root@localhost:/root>
   ```

4. Keep this window to your target open to observe possible errors when you run the program.

You are now ready to connect Workbench on your host to the agent on the target.

If you want to build usermode-agent for a Wind River Linux target, but in an external location, see *3.11.2 Building usermode-agent at External Locations*, p.85.

3.6 Testing Target Connections

Once you have configured a target connection and started **usermode-agent** on the target, you can connect to the target from Workbench.

To make and verify the connection:

1. Select the target connection in the Remote Systems view.
2. Click the green **Connect '***Connection_name'* icon in the Remote Systems toolbar.
3. Expand the target connection in the Remote Systems view so you can find the **Processes** node.
4. Find the list of processes the target is running. One of these will be **usermode-agent**.

Later, when you have completed running and/or debugging your application, you can close the connection:

1. Highlight it in the Remote Systems view.
2. Click the red **Disconnect** '*Connection_name'* icon.

3.7 Using the Remote System Explorer

The Remote System Explorer (RSE) allows you to:

- Browse a remote target's file system
- Browse your host's target system
- Copy files from directory to directory within the target
- Issue remote shell commands to your target

Once you have opened the Remote System Explorer and used it as described in the following sections, you close it by right-clicking the **Sftp Files** node and choosing **Disconnect**. (See Step 2 in *Browsing the Target's File System*, p.68.)

When you re-open **Stfp Files** or **Root**, Workbench reconnects you to the target and you can continue browsing, copying files, and issuing remote shell commands.

NOTE: This feature requires ftp and ssh on the host, which small file systems currently do not by default include.

For another approach to working with the target file system from within Workbench, see *5.5 Working with Packages and Target File Systems*, p.121.

Browsing the Target's File System

To browse the target's file system using RSE:

1. Start the usermode-agent on the target, in background mode, as explained in *3.5 Running Usermode Agent on Non-Emulated Targets*, p.65.

   It should already be running if you have achieved a target connection.
2. Expand the remote system connection in the Remote Systems view so you can see the nodes **Sftp Files** and, within that, **Root**.
3. Click **Root** to open it.
4. Enter the username (**root**) and password (**root**) in the dialog box.
5. Browse the target's file system under **Root**.

Copying Files Within the Target File System

To copy files within your target's file system:

1. Browse to the file under the **Root**.
2. Right-click the filename and select **Copy**.
3. Browse to where you want to place the file and select **Paste**.

Issuing Remote Shell Commands to Your Target

To issue remote shell commands:

1. Right-click **Ssh Shells** under the connection in the Remote Systems view.
2. Select **Launch shell**.

   A Remote Shell view opens in Workbench, with the name of the connection.

3. In the **Command** text box in the Remote Shell view, enter a command, such as **uname -a**, and press **ENTER**.

   Command output appears in the Remote Shell's terminal window area.

4. Use the up or down arrows at the end of the Command field to retrieve previously issued commands.

You can have multiple such shells open at the same time, which would be useful if you are tracking disparate tasks.

3.8 Using the One-Step Launch Configuration

You can tailor the default launch configuration and then use it at any time as a one-step process to connect to your target and launch the program.

NOTE: This section assumes that your target is running the usermode-agent as described in *3.5 Running Usermode Agent on Non-Emulated Targets*, p.65.

3.8.1 Accepting the Automatic Debug Launch Configuration

Workbench automatically specifies certain parameters for the default launch configuration for Linux application projects.

When you select the application's Binary node in the project folder and choose **Debug Process on Target**, the automatic settings include:

- Adding the application image to the File Copy tab for transferring to **/root** on the target.
- Adding the required object path mappings to the connection properties.
- Updating the **Exec Path on Target** attribute based on these default mappings.
- Creating an automatic breakpoint at **main( )**.

These settings result in automatically transferring the application image to the target (which is already running **usermode-agent**), running the application, and breaking at **main( )**.

In addition:

- You may modify the file transfer to transfer the file to a location other than **/root** on the target. This change persists in later uses of the launch configuration.
- You may launch the same image multiple times without the image being transferred each time.
- You can suppress the transfer completely by disabling the file transfer. (You must disable the transfer, not just remove it.)

This automation and flexibility are associated with a few limitations, such as:

- If you rebuild the image, the transfer will happen again: there is currently no mechanism for recognizing that the rebuild might have resulted in the same image.
- The image transfer fails if any process created from it is still running on the target.

3.8.2 Modifying the Default Launch Configuration

If the automatic default configuration is inappropriate for some reason, you may modify the launch configuration as follows:

1. Right-click the application that you want to run on the target.
2. Select **Run Process on Target** to begin to create a new debug launch configuration.
3. On the left, select **Process on Target**.
4. Click the **New launch configuration** icon in the toolbar.

   A **Create, manage, and run configurations** window opens, with an area for creating a launch configuration in the right portion of the dialog box.
5. In the **Name** field, accept the supplied name or enter a new name, such as **penguin**, for the launch configuration.
6. In the **Exec Path on Target** field, browse to *your executable*.

   For example:

   - For QEMU:
     workspace/**penguin/arm_versatile_926ejs-glibc_std.armv5tel_fvp/penguin/Debug/penguin.out**

- For the Power PC:
  /opt/windriver/ppc85xx/hello_Linux/*board-fs-CPU***/hello_linux/Debug/hello_Linux.out**

7. In the **Working Directory** field, browse to the *directory containing your executable*. For example:
   - For QEMU:
     workspace/**penguin/arm_versatile_926ejs-glibc_std.armv5tel_fvp/penguin/Debug**
   - For the Power PC:
     /opt/windriver/ppc85xx/hello_Linux/*board-fs-CPU***/hello_linux/Debug**

3.8.3 Using the Launch Configuration to Connect to the Target

If you have modified the default launch configuration as explained in the previous section, you can use it to connect to the target, as follows:

1. Confirm that you are still looking at the **Create, manage, and run configurations** dialog that you have just modified.
2. If **Connect** is *not* grayed-out, click **Connect**.
3. If **Connect** is grayed-out, click **Run**.
4. Click the arrow to the left of the connection to display entries for the target architecture and processes.

Workbench is connected to the agent on the target, and your build results are redirected to the target. You are now ready to use Workbench to run and debug applications on the target.

For more detailed information on launch configurations, see *17.2 Creating a Launch Configuration*, p.266.

3.9 Debugging an Application on the Target

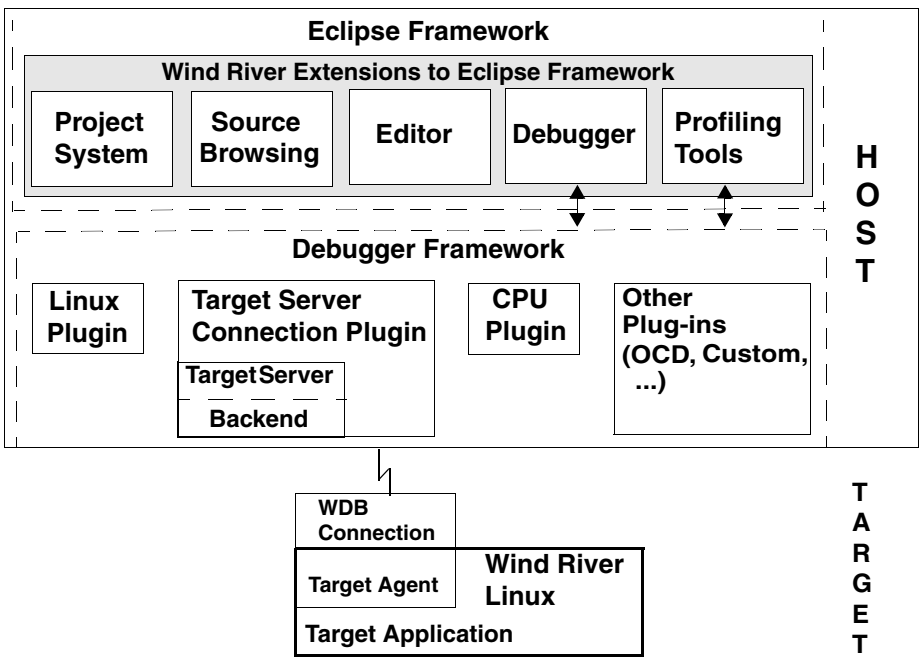
This section briefly explains how to:

- Recognize the parts of the Workbench debugging environment
- Run a program on a target until it reaches the first breakpoint
- Use the device debug perspective
- Step through code and examine memory
- Set and modify breakpoints

3.9.1 The Workbench Debugging Environment

Figure 3-2 illustrates how Wind River extends the Eclipse framework and adds plug-ins to form the debugger framework. The debugger connects to the target agent on the target through the target server on the Workbench host.

Figure 3-2 **Workbench Host and Target Relationship**



3.9.2 Running on the Target up to a Breakpoint

The following procedure automatically creates a launch configuration for you, which you can access by right-clicking in the Debug view.

You may, however, connect to the target from the Remote Systems view or the Project Explorer instead of using a launch configuration.

To run your program on the target and start to debug it:

1. If your Debug view is not still open from the previous procedure:
   a. Select **Run > Debug Last Launched** to open it.
   b. Select your launch on the left to display the launch configuration.
2. Click **Debug** to connect to your target and launch the program.

   Workbench:

   - Builds the program.
   - Connects to the target if not already connected.
   - Switches to the Device Debug perspective.
   - Downloads the program and runs it on the target (in this case, the program is run as an ordinary Linux application program). The program executes up to **main( )** and breaks.
   - Opens the source file and highlights the location of the program counter where execution has stopped in the **main** routine.

NOTE: Instead of using the menu selections, you could use **F11** or click the debug icon in the main menu bar.

3. Confirm that the program has stopped at a breakpoint by looking at the Debug view in the upper-right corner of your window.
4. When you connect to your target and launch the program, the connection displays in the Remote Systems view with **[connected]** beside it. The type of target and version of Linux that is running is displayed under the connection definition. You can expand this node to display any tasks that are running. Scroll down and find the stopped **ball.out** process. It will have the same process ID shown with the stopped process in the Debug view.
5. The Workbench window now displays the Device Debug perspective with the source file open and the line highlighted corresponding to the point at which the program stopped.

NOTE: You can connect to the target from the Remote Systems view or the Project Explorer instead of using a launch configuration. Connecting this way automatically creates a launch configuration for you, which you can access by right-clicking in the Debug view.

Now that you have created a launch configuration, you do not have to go through these steps each time you want to test your application—the connect, run, and attach debugger steps are all pre-configured. You can use a launch configuration in either run or debug mode. You can click any of the following:

- The **Run** button in the toolbar to run the most recently launched application.
- The **Debug** button in the toolbar to debug the most recently launched application.
- The arrow next to the **Run** button in the toolbar to select an application to run from among your most recent launches.
- The arrow next to the **Debug** button in the toolbar to select an application to debug from among your most recent launches.

Refer to *17. Launching Programs* for more information on the run and debug modes of launch configuration.

3.9.3 Using the Device Debug Perspective

The default Device Debug perspective includes views suited to executing and debugging a program, including the Project Explorer and Remote Systems views as before, but replacing the Outline view with the following elements:

- The Debug view
- The Breakpoints view
- A tabbed notebook that includes the Variables, Expressions, Registers, and Memory views

Notice that the new perspective is present in the shortcut bar at the top right edge of the Workbench window, where a button for **Device Debug perspective** has been added to the buttons for **Open a Perspective** and the **Application Development perspective**. Switch between the two perspectives by clicking their buttons.

As with the Application Development perspective, the views in the Device Debug perspective can be repositioned to suit your needs.

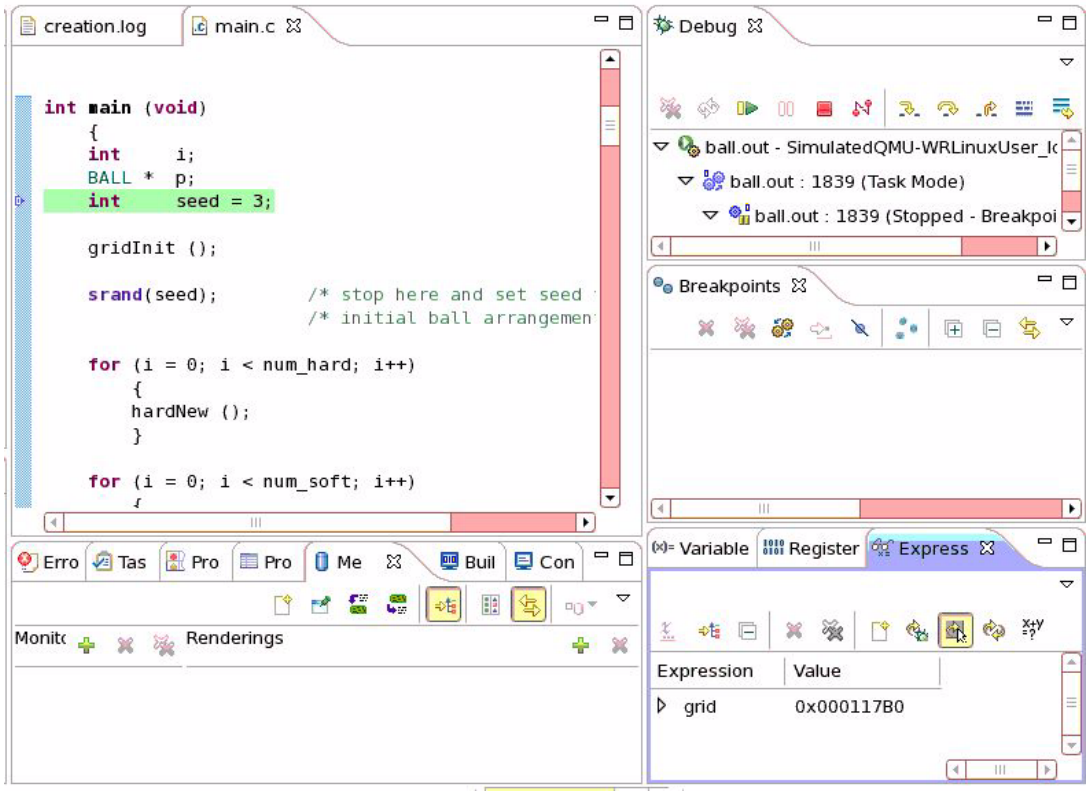
The action of the ball sample program is viewed in the Memory view**.**

To set up the Device Debug perspective to match this tutorial, do the following:

1. Select **Window > Show View > Memory**.
2. Click the Memory tab at the bottom right, drag it over the Build Console view (wait for an icon of a set of stacked folders to appear at the cursor as you move it), then drop the view.
3. Select **Window > Show View > Expressions**.
4. Right-click in an empty area below the column label Expression, select **Add Watch Expression**, enter the grid global variable **grid** under **Name**, and press **ENTER**.

   The memory address of the **grid** global variable appears in the Value column. This address can vary from one session to another if something changes, for example, if you compile with the GNU compiler instead of the Wind River compiler.

The Device Debug perspective now appears approximately as shown:



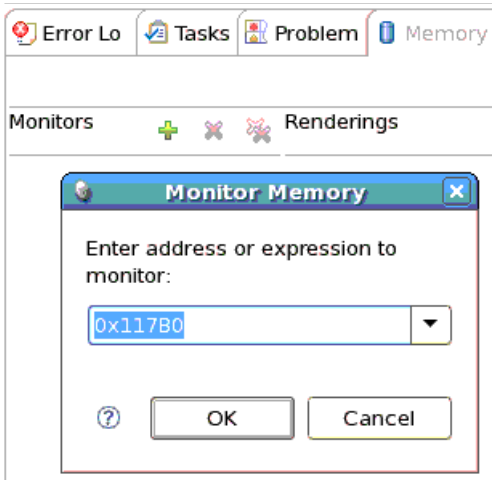
3.9.4 Stepping Through Code

This procedure takes you through the initialization of the Grid array in the ball sample program.

1. Press **F6** (or use the **Step Over** icon in the Debug view) twice to step from the entry point of **main( )** to the call to **srand( )**.

   Using **F6** twice causes Workbench to step over and complete the execution of **gridInit( )**. (All the run controls are available on the Run menu, and also as toolbar icons in the Debug window.)

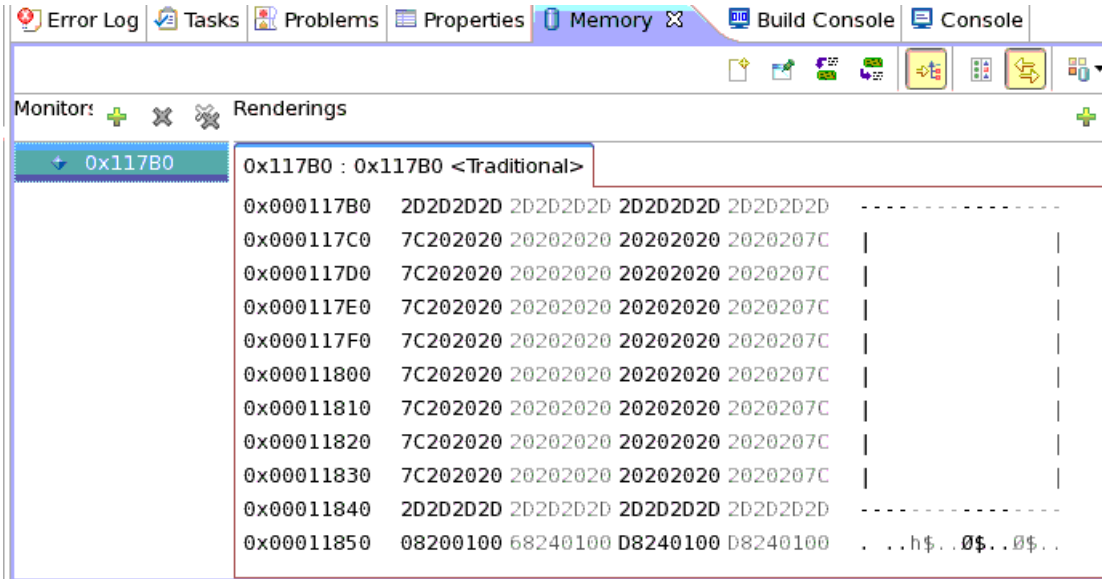
2. In the Memory view, click the Add Memory Monitor icon in the Monitors toolbar, enter the value (address) of the **grid** array from Step 4 in *3.9.3 Using the Device Debug Perspective*, p.74, and click OK.



3. Adjust the Memory view to show the box outlining the grid on which the balls will bounce:

   a. Right-click in the Renderings column and select **Cell Size > 8 bytes**.

   b. Drag the right or left borders of the Memory view to make it exactly wide enough, and drag the top and bottom borders to make it high enough for the box in the text area of the window to appear correctly as shown below.



If the box does not appear, make sure the address you entered in the Memory view is that of the **grid** global variable.

3.9.5 Setting and Running to a Breakpoint

The box may be empty now, but as the program runs, characters representing different types of balls (two zeros, two @ signs, and two asterisks) appear in the box, bounce around, and collide with each other and with the walls.

The following procedure explains how to set a breakpoint and tell the application to run until the breakpoint is reached:

1. Scroll down in **main.c** past the three initialization **for** loops and set a breakpoint at the following **while** statement, by double-clicking in the gutter next to the **while** statement:

```
    /*
     * To watch the balls move, display global "grid" in the
     * Memory window, and place a breakpoint with property
     * "Continue on Break" enabled at either the next statement
     * or at the "for" statement for a more detailed view.
     */

    while (! finished)
        {
        for (p = pMovableBalls; p != NULL ; p = p->pNextMovable)
            {
            p->pBallMove (p);
            }
        }

    return 0;
    }
```

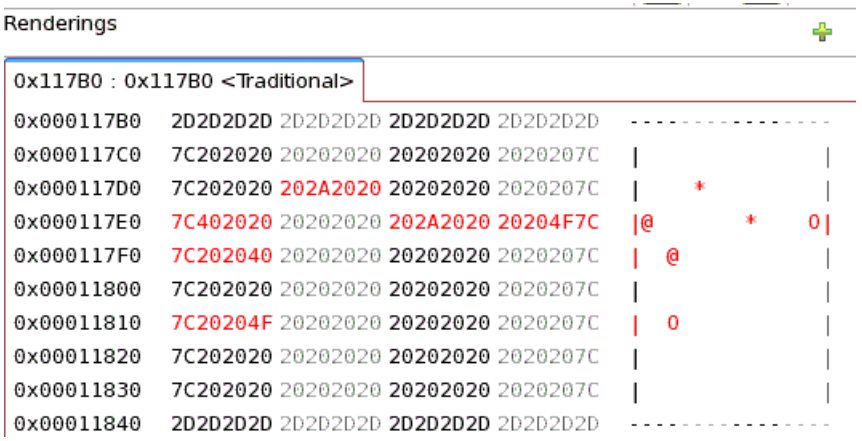
A blue dot appears in the gutter.

The Breakpoints view displays the module and line number of the breakpoint:

NOTE: If no comment appears above the **while** statement, click the **+** symbol in the gutter next to it to unfold the comment. You can click the **+** symbol again when you are done reading the comment to fold it away.

2. Press **F8** (or click the **Resume** icon in the Debug view's toolbar) to run to the breakpoint.

   Execution stops each time it hits the breakpoint. The Memory view changes each time you resume execution. The highlighting in the following figure indicates changes after the first refresh.



3.9.6 Modifying the Breakpoint to Execute Continuously

Next, change the behavior of the breakpoint so that the application does not stop executing at each break but instead refreshes the display:

1. Right-click the breakpoint in the gutter, or in the Breakpoints view, and select **Breakpoint Properties**.

   The Line Breakpoint Properties dialog appears.

2. Click the **Continue on Break** check box.
3. Click **OK**.
4. Press **F8** or click the **Resume** button to watch the balls bounce in the Memory view.

3.9.7 Sending Signals to Processes

The Remote Systems view displays all available processes.

To send a POSIX-compliant signal to a process:

1. Right-click the process and select **Send Signal**.
2. Select a signal from the list and click **OK**.

   Workbench sends the signal to the selected process.

For example, send a SIGINT or SIGTERM to terminate a process. If these are not successful (perhaps the process is stuck in a system call), send it a SIGKILL.

Note that the Sun Java Virtual Machine (JVM) uses SIGQUIT to create a stack dump on the terminal. Sending SIGQUIT to any running JVM dumps the stack trace of all active threads to the terminal associated with the JVM.

For more details, see *18.2.8 Using the Debug View to Send Signals to Processes*, p.296.

3.9.8 Terminating the Process

When you are done:

1. Click the Terminate icon in the Debug view's toolbar. (It looks like a red box.)
2. Click the **Remove All Terminated Launches** button (two gray X's) to remove the terminated launch.

3.10 Customizing Builds with Layers and Templates

Workbench's Platform Project wizard allows you to select features to customize builds, for purposes such as:

- Adding packages
- Creating new BSPs
- Ensuring that custom builds can be easily repeated
- Completely separating custom code from the installed Wind River development environment

This customizing facility is implemented as templates and layers. These can sometimes be used interchangeably, but they are designed for different purposes.

- *Templates*, suitable for small or discrete changes, typically are incorporated into a project at the end of the configuration process.
- *Layers* are appropriate for adding new packages and making significant changes to a BSP's templates (for example).

3.10.1 Adding Templates to a Project

To add templates to a platform project:

1. Take the initial steps in creating the project, to the point where you have named it and have reached the Configure Options screen. In other words, follow up to Step 5 in *3.2 Creating a Wind River Linux Platform Project*, p.54.
2. Click **Advanced > >**.
3. Click **Add** to the right of the **Templates** box.
4. Select boxes to specify which templates you want from the resulting list.
5. Click **OK**.

See Table 3-3 for a description of the supplied templates.

The following supplied templates are in *installDir*/**wrlinux-2.0/wrlinux/templates**, in the directories **extra** and **feature**.

Table 3-3 **Supplied Project Configuration Templates**

Template	Purpose
extra/wrsnmp	Enable Wind River SNMP and the SNMP Agent
feature/acpi	Incorporate and start the logging daemon on kernel-acpi enabled boards with cgl and std file systems
feature/busybox	Add busybox package and set up the config file in **busybox/busybox/config**
feature/debug	Add usermode-agent to small-filesystem projects
feature/directfbfusion	Allow multiple DirectFB applications to run at the same time within a session
feature/glibc_discrete	Replace busybox with a set of full-featured, discrete utilities such as in glibc_standard
feature_glibc_small_extras	Replace busybox with a set of full-featured, discrete utilities such as in glibc_small
feature/ipmi	Add IPMI package and enable support for IPMI controllers.
feature/prelink	Modifies ELF shared libraries and executables, to require fewer relocations to be resolved at runtime

Table 3-3 **Supplied Project Configuration Templates**

Template	Purpose
feature/uclibc	Replace glibc with the uclibc C library for MIPS and ARM targets with one or more uclibc rootfs names
feature/unionfs	Allow booting from CDROM for BSPs with CDROMs

To create and add custom templates to a project, see *Wind River Linux Platforms User's Guide, 2.0*. Although you may place custom templates anywhere, they typically reside outside of both the development and build environments.

Custom templates are processed last, after all other templates. Because of this, they are especially useful for:

- Kernel configuration file fragments that must come at the end of a kernel configuration file
- Additions to file system configuration files, for example, networking configuration files in **/etc/sysconfig**, that may have to override default values
- The **pkglist.add** and **pkglist.remove** files that may have to override previous package list files

You also may add a one-line kernel configuration to be processed last.

3.10.2 Adding Layers to a Project

To add layers to a platform project:

1. Take the initial steps in creating the project, to the point where you have named it and have reached the Configure Options screen: follow up to Step 5 in *3.2 Creating a Wind River Linux Platform Project*, p.54.
2. Click **Advanced**.
3. Click **Add** to the right of the **Layers** box.
4. Browse to the directory containing a layer to include in the project.

   Because part of Wind River Linux is implemented using layers, the Workbench *installDir* contains some, such as the kernel implementation in the layer **wrlinux-2.0/layers/wrll-linux-2.6.21**. However, the layers you choose to add to the project come from directories other than *installDir*.
5. Click **OK**.

Layers make it easy to package, locate, and review a set of changes, allowing you to back out undesirable changes, and to share your changes in a coordinated fashion. You could, for example, add packages, remove other packages, and add and remove different kernel features with a single layer. You could then distribute your layer to a group of developers or simply make it available over the LAN. Other developers could then easily include or exclude your changes with a single configure command switch

You can create layers automatically from a build environment you have modified, or you can create them manually as described in *Wind River Linux Platforms User's Guide, 2.0*.

Custom layers are processed differently from custom templates. During processing, the system looks first to the custom layer in an attempt to find the template, package, or file it is looking for. Thus, custom layers can be used to add templates, packages, or files, or to override the templates, packages or files within the development environment. For instance:

- A new package added to a custom layer will be added to the build system.
- A template within a custom layer will be used instead of the identically named standard template with the development environment.

Although custom layers can be placed anywhere, they typically reside outside of both the development and build environments.

3.11 Working with External Locations

This section explains how to create projects and build **usermode-agent** at external locations.

External locations are external to your workspace directory.

3.11.1 Creating User-Defined Projects at External Locations

In the example of the **ball** program, you imported existing files into a project in your workspace. In some cases, you may prefer to use the files in their existing location rather than, for example, maintaining copies of them in two separate locations.

You can do this by creating a user-defined project at the location of the source files. The project still appears in the Project Explorer like any project in your workspace, but the actual files, including Workbench metafiles (**.project** and **.wrproject**) are located in a location external to your workspace directory.

3.11.2 Building usermode-agent at External Locations

The following example shows how to build the **usermode-agent** at an external location, using the source files that come with the Workbench installation.

1. Create a new user-defined project by right-clicking in the Project Explorer and selecting **New > User-Defined Project**, so that the makefile in the supplied source will be used.
2. If prompted, select your target operating system.
3. In the **New User-Defined Project** dialog box, assign a project name such as **usermode-agent**.
4. Select **Create project at external location**.
5. Click **Browse** and locate the **usermode-agent** source files located in *installDir*/**linux-2.x/usermode-agent/1.1/src**. Click **OK**, then **Next**.
6. In the **Project Structure** dialog box, click **Next**.
7. In the **Build Support** dialog box, select **User-defined build**. Leave the **Build command** text as it is for now. Click **Finish**.
8. Expand your new project in the **Project Explorer**. Notice that the project contains the **usermode-agent** source files as well as the project metafiles. Scroll to the **README** file and double-click it to open it in the Editor.
9. In the **README**, find the appropriate **make** text for your target architecture and copy it. If you are building **usermode-agent** for a Wind River Linux target, refer to *3.5 Running Usermode Agent on Non-Emulated Targets*, p.65.
10. In the **Project Explorer**, right-click the project folder and select **Properties**. Select **Build Properties,** and in the Build Support tab, replace the **Build command** text with the **make** text you copied from the **README** file, substituting any values as appropriate for your environment.
11. You can now build the project. The resulting binary will appear in the **bin/***arch* subdirectory of the **src** directory.

4

Kernel Debugging (Kernel Mode)

4.1 Introduction 88

4.2 Background Steps and Concepts 89

4.3 Preparing the Target for KGDB 93

4.4 Connecting with KGDB from Workbench 95

4.5 Attaching and Debugging the Kernel from Workbench 99

4.6 Debugging User-Supplied Kernel Modules 101

4.7 Using QEMU for KGDB Kernel Debugging 110

4.8 Enabling and Disabling KGDB in the Kernel 113

4.1 Introduction

Wind River Workbench provides source-level debugging for Linux 2.6 kernels that support the Kernel GNU Debugger (KGDB).

NOTE: If you are *not using a Wind River Linux platform*, contact Wind River support about configuring your kernel for symbolic debugging with Workbench.

With KGDB installed, you can debug the kernel much as you debug applications, including stepping through code, setting breakpoints, and examining variables. Kernel mode debugging between Workbench and the target takes place over a serial protocol connection which may be over a serial line or Ethernet.

NOTE: With Wind River Linux, kernel debugging is enabled by default and you only have to turn it off when going to production.

Wind River is a contributor to the KGDB open source project. Basic community information on KGDB can be found at **http://sourceforge.net/projects/kgdb/**.

Wind River has added several features to the mainline KGDB, including:

- Functionality that is tested on all supported boards
- Support for CPU auto-detection
- Additional flags and test features added to the KGDB agent.

See *Wind River Linux Platform User's Guide* for information about using the command line to:

- Debug with KGDB
- Enable and disable KGDB
- Build your own cross-architecture gdb

4.2 Background Steps and Concepts

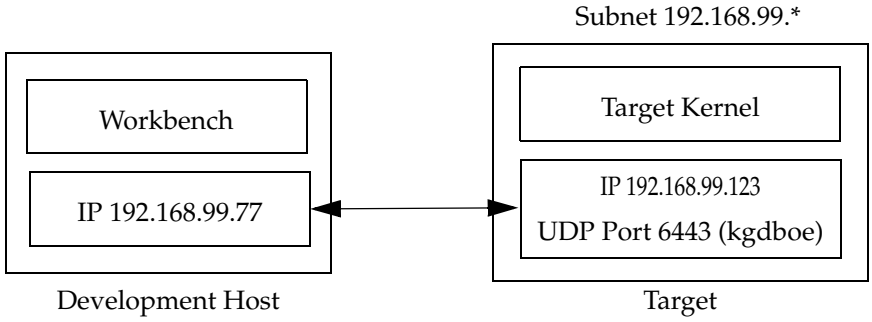
This section briefly describes how to set up KGDB to work over subnets using agent-proxy. It describes how to mount the target file system. And it explains how to disable KGDB, which by default is enabled.

4.2.1 Setting Up for kgdboe Between a Target and a Development Host

KGDB over Ethernet is designed to work within single subnets, since it is based on unicast UDP packets.

The connection is from the host over UDP to the target, using the standard port 6443 for KGDB, as sketched in the following figure:

Figure 4-1 **Connecting from the Host to the Target for KGDB**



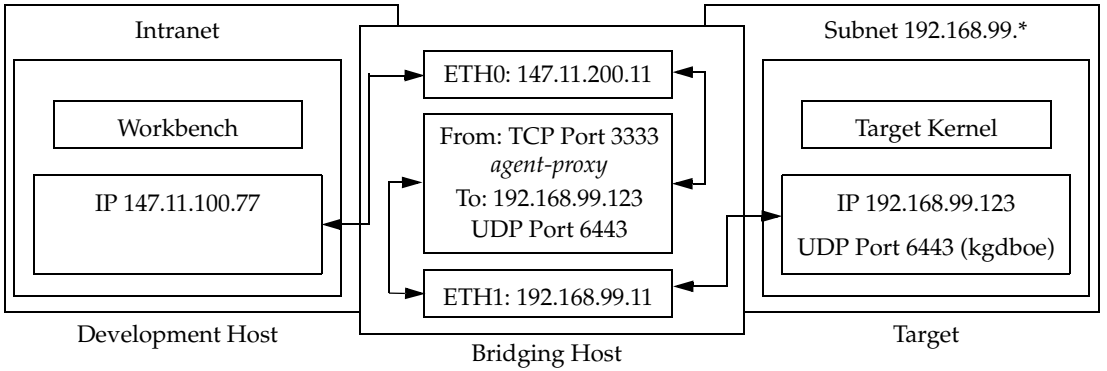
4.2.2 Using agent-proxy to Cross Subnet Boundaries for kgdboe

Wind River Linux provides a utility that can convert TCP connections on an intranet to UDP connections in a subnet. If a host is available that sits on both the intranet and the subnet, the utility **agent-proxy** can provide this required access. The agent-proxy is in the **host-tools/bin** directory, which you would copy to the bridging host.

Example instructions for using agent-proxy:

1. Log into the bridging host. In this example, the bridging host has one Ethernet connection on the intranet (147.11.200.11), and a second Ethernet connection on the target's subnet (192.168.29.11). (See Figure 4-2.)

Figure 4-2 **Using agent-proxy to Bridge Subnets**



2. Start a copy of agent-proxy. This will connect the TCP port 3333 to the UDP port 6443 from 192.168.99.123

```
$ ./agent-proxy 3333 192.168.99.123 udp:6443
```

3. Log into the target (192.168.99.123), and enter this command. This will start the kgdboe agent, connected to the bridging host.

```
root@target:/>modprobe kgdboe kgdboe=@192.168.99.123/,@192.168.29.11/
```

4. Log into the development host on the intranet.
5. Start Workbench, and create a kgdboe connection with these parameters.

```
TRANSPORT=TCP
CPU=default
IP=147.11.200.11
Port=3333
```

This has Workbench connect to the TCP port 3333 on 147.11.200.11, and treat it like a normal KGDB connection.

NOTE: You can have any number of agent-proxy connections running on the bridging host, one for each target within the subnet. Just assign a different TCP port, for example 3334, 3335, 3336, and so forth.

If you run **agent-proxy** without any parameters, you will get a list of its commands and abilities, as shown here:

```
$ wrlinux/host-tools/bin/agent-proxy
agent-proxy version 1.6
Usage:
 agentproxy <[udp:][virt IP:]local port> <remote host> <[udp:]remote
port>
   For udp wdb agent proxy example for default ports
    agentproxy udp:0x1001 10.0.0.2 udp:0x4321
    agentproxy udp:0x1002 remotehost2 udp:0x4321
   For KGDB udp target to udp host with default g
    agentproxy udp:3331 10.0.0.2 udp:6443
   For KGDB udp target to tcp host with default g
    agentproxy 3332 10.0.0.3 udp:6443
   Also you can bind to a local vitual interface IE:
      agentproxy udp:47.1.1.2:0x4321 10.0.0.2 udp:0x4321
      agentproxy udp:47.1.1.3:0x4321 10.0.0.3 udp:0x4321
      agentproxy udp:47.1.1.3:6443 10.0.0.3 udp:6443
      agentproxy 47.1.1.3:44444 10.0.0.3 44444
```

The KGDB over serial connection does not have this limitation. You will need either physical serial cable that can make the connection or, alternatively, you can set up a Telnet port repeater that connects the serial connection to a Telnet port on the intranet.

4.2.3 Mounting the Target's File System via NFS

If you already have the NFS image mounted to your target's file system, skip this procedure.

To allow easy kernel module transfers from the host to the target, especially if your target is a NFS mounted via **pxeboot**:

1. Open a console on your host.
2. Change to super-user mode:

   ```
   $ su
   ```

3. Create a mount directory for the target rootfs.

   ```
   $ mkdir /target
   $ chmod a+rw /target
   ```

4. Mount this directory. In this example, the many lab boards get their root file systems from NFS directories on a shared host.

   ```
   $mount -t nfs ala-tuxlab://export/pxeboot/my_board/rootfs /target
   ```

4.2.4 Kernels: Enabling KGDB versus Debug Builds

The KGDB feature is a Linux kernel configuration option, as opposed to compiling the kernel using the debug flags. By default, KGDB support is enabled in the pre-built and configured Wind River Linux kernels.

You can configure a platform project with **--enable-debug**, but this has no effect on KGDB and only applies to the package builds. The pre-built or generated **vmlinux** file contains all the required debug information required for the kernel.

See *4.8 Enabling and Disabling KGDB in the Kernel* , p.113, to disable KGDB.

4.2.5 KGDB and Safe Areas when Debugging the Kernel

Due to the nature of KGDB debugging, not all sections of the kernel can be safely or reliably debugged. In kgdboe for example, it is impossible to debug the Ethernet driver itself or any of its related services. The kgdb8560 serial interface option in general has fewer dependencies, but it still has sections where it cannot go, and it runs slower than the Ethernet connection. The On Chip Debugger (OCD) has the fewest restrictions on where is can debug, since it does not rely on kernel services at all, but does require additional hardware to support it.

Also, there will be different behavior in parts of the kernel depending on the specific architecture or configuration. If a common breakpoint location (for example **do_fork**) does not seem to work on a particular installation, note it and find an alternative break location.

While these restrictions may discourage some people from employing interactive kernel debugging, this feature does provide tremendous advantages over simple using **printk** in spite of these restrictions.

Here are some of the sections of the kernel code to specifically avoid:

- **Spin locks**. The Ethernet driver has several spin locks, and if you attempt to step over one that is needed, then the connection will lock up.
- **Printk**. This routine also has spin locks that will freeze the KGDB connection.
- **Simultaneous usermode-agent** connections. The two debugging connections indirectly and unavoidably share some spin locks.

4.3 Preparing the Target for KGDB

You need to start the KGDB agent on the target before you can make any connection from Workbench.

You can use two connection methods to connect to KGDB, the first over the Ethernet via UDP, and the second using a serial connection either directly to a COMM port (or otherwise over Telnet). The Ethernet KGDB connections are typically much faster than serial connections.

4.3.1 Option 1: Using a UDP Connection on a Non-PCD Target

On the target with the KGDB enabled kernel, find the IP address of the target and the IP address of the host where you'll be running gdb/Workbench.

The **kgdboe** module usage syntax follows. It is very rare that you will need to enter the source port or the host mac address.

```
kgdboe=[src-port]@[src-ip]/[dev],[hst-port]@hst-ip/[hst-macaddr]
```

It is recommend to explicitly call out the Ethernet port since this guarantees that kgdboe will used the correct port (kgdboe does not know which port the device was booted with). Since more and more boards have multiple Ethernet ports, the default of **eth0** may not be correct.

```
Target_# modprobe kgdboe kgdboe=@/eth2,@192.168.29.107
```

NOTE: To underline this point, if this board has **multiple Ethernet ports** and/or the port you want is not the default, you *must* explicitly select that alternate Ethernet port. You can use **ipconfig** to identify the proper port to use.

If you have problems connecting to KGDB, double check which port is configured and connected to the network If for example the **modprobe** commands takes a long time to return, it may be accessing or defaulting to a port that is not connected, and the only other sign of error is that kgdboe connections do not succeed.

Here are some common short cuts for starting kgdboe:

- This version assumes the default Ethernet port, for example **eth0**:

  ```
  target_# modprobe kgdboe kgdboe=@/,@192.168.29.107/
  ```

- This version explicitly calls out the target's IP address:

  ```
  target_# modprobe kgdboe kgdboe=@192.168.29.96/,@192.168.29.107/
  ```

You can now skip to *4.4 Connecting with KGDB from Workbench*, p.95.

4.3.2 Option 2: Using a UDP Connection on a PCD Product

In the PCD product, the kgdboe module is statically and automatically linked into the kernel.

There are two ways to configure the **kgdboe**, as described here.

Statically Configuring KGDB-OE for PCD (uclibc_small+small)

The first method is to add the **kgdboe** configuration into the boot command. You must know the IP address of both the target and the host at boot time. You can immediately connect with KGDB anytime after the boot completes.

For example, if you had:

- Target IP: 10.1.1.12
- Host IP: 10.1.1.111

On the boot line you would need

```
kgdboe=@10.1.1.12/,@10.1.1.111/
```

So the boot line would look like:

```
boot root=/dev/nfs rw console=ttyS0,115200
kgdboe=@10.1.1.12/,@10.1.1.111/ nfsroot=10.1.1.111:/rootfs ip=10.1.1.12::
10.1.1.1:255.255.255.0:mainstone:eth0:off mem=128M
```

And during the boot you would see:

```
kgdboe: device eth0 not up yet, forcing it
eth0: link down
eth0: link up, 100Mbps, full-duplex, lpa 0x45E1
kgdboe: debugging over ethernet enabled
kgdb: Defering I/O setup to kernel module.
```

You can now skip to *4.4 Connecting with KGDB from Workbench*, p.95.

Dynamically Configuring KGDB-OE for PCD (uclibc_small+small)

The kernel now also supports the new dynamic kgdboe configuration when used as a kernel built-in. You do:

```
$ echo @/,@10.1.1.111/ > /proc/sys/kgdboe
```

To see the current configuration, enter:

```
$ cat /proc/sys/kgdboe
```

NOTE: Since the device has booted, you do not need to specify its IP address.

You can now skip to *4.4 Connecting with KGDB from Workbench*, p.95.

4.3.3 Option 3: Using a Serial/Telnet Connection

On the target with the KGDB enabled kernel, find out the IP address of the target and the IP address of the host where you'll be running gdb or Workbench.

The **kgdb-8250** module usage syntax is:

```
8250_kgdb kgdb8250=com-port,baud-rate
```

This example is for a COM1 connection at 115200 baud:

```
target_# modprobe 8250_kgdb kgdb8250=0,115200
```

This example is for a COM2 connection at 9600 baud:

```
target_# modprobe 8250_kgdb kgdb8250=1,9600
```

Do not use a port that has a console attached to it. You may make a connection, but it will not otherwise work, because the console will attempt to interpret the KGDB data as typed commands.

Also, for 9600 baud connections, you may want to reduce the KGDB watch options to achieve reasonable performance.

4.4 Connecting with KGDB from Workbench

This section guides you in creating the Target Connection to the target for KGDB from Workbench. This requires that the KGDB agent is already running on the target, as described in *4.3 Preparing the Target for KGDB*, p.93

4.4.1 Option 1: Making an Ethernet Connection

1. Start Workbench on your host.
2. Create a new target connection.

3. Select the connection type **Linux KGDB**, and click **Next**.
4. Select the connection option **Linux KGDB via Ethernet**, and click **Next**.
5. Select **UDP** as the transport.

NOTE: If you are using the agent-proxy, make this a TCP connection, as described in *4.2.2 Using agent-proxy to Cross Subnet Boundaries for kgdboe*, p.89.

6. Use the **Default from Target** CPU default.

NOTE: Select the Arch only if necessary (for Pentium targets, use **Pentium4** instead of **ia32**). Use the new architecture families when explicitly selecting the target CPU type.

7. Enter the target's IP address.

   The IP address must match the setting from *4.3.1 Option 1: Using a UDP Connection on a Non-PCD Target*, p.93.

8. Use port 6443.

   This port is reserved for KGDB connections. If you are using the agent-proxy, select the respective agent-proxy port number, as described earlier.

9. Click **Next**.
10. Select the **Target OS Linux 2.6** if it is not already set.
11. For the **Kernel Image**, point to your kernel's **vmlinux**.

    ```
    $ mybuild/export/*-vmlinux-*
    ```

    If you have built the kernel, the physical location of the new symbol file is here:

    ```
    $ mybuild/build/Linux-2.6.14/vmlinux
    ```

NOTE: Do not select the kernel image itself (for example the **bzImage** file).

12. Click **Next**.
13. For the **Object Path Mappings** screen, click **Add**, set the following values, and click **OK**:
    - **Target Path =** *(leave empty)*
    - **Host Path = /target/**
14. Select **Next** through the following dialog boxes, then press **Done**.

This may take some time to load all the kernel symbols, on the order of 20 to 40 seconds. When it completes, there will be a red "**S**" on the **vmlinux** element in the target's connection tree.

NOTE: Make sure that the target has started the module kgdboe before opening the connection in Workbench, else you will get a timeout and an connection error summary.

When the connection is successful, the Remote Systems view displays the connected message, and the kernel is shown as **Stopped**.

Proceed to *4.5 Attaching and Debugging the Kernel from Workbench*, p.99.

If the connection failed, go to *4.4.4 When a KGDB Connection Fails*, p.98.

4.4.2 Option 2: Making a Telnet Connection

A port selector device can aggregate multiple serial port connections into telnet connections that can be made available across the network. The IP address for this telnet virtual connection would be the one for this port selector device, and the port number would be the one for the port of that target.

For example, if your multiplexer **digiswitch_1** was at 192.168.29.2 and the telnet serial port for the target was 2011, then the IP address would be 192.168.29.2, and the port would be 2011.

To make a KGDB telnet connection:

1. Start Workbench on your host.
2. Create a new target connection.
3. Select the connection type **Linux KGDB**, and click **Next**.
4. Select the connection option **Linux KGDB via Terminal Server**, and click **Next**.
5. Select **TCP** as the transport.
6. Use the **Default from Target** CPU default.

NOTE: Select the Arch only if necessary (for Pentium targets, use **Pentium4** instead of **ia32**).

7. Enter the target's IP address.

The IP address must match device providing the Telnet connection.

8. Enter the Telnet port.
9. Click **Next**.

Proceed to *4.5 Attaching and Debugging the Kernel from Workbench*, p.99.

If the connection failed, go to *4.4.4 When a KGDB Connection Fails*, p.98.

4.4.3 Option 3: Making a Serial Cable Connection

To make a serial cable connection:

1. Start Workbench on your host.
2. Create a new target connection.
3. Select the connection type **Linux KGDB**, and click **Next**.
4. Select the connection option **Linux KGDB via RS-232**, and click **Next**.
5. Select the Baud rate**,** Data Bits, Parity, Stop Bits, and Flow control appropriate to the connection. Note that the baud rate must match the setting from *4.3.3 Option 3: Using a Serial/Telnet Connection*, p.95.
6. Click **Next**.

Proceed to *4.5 Attaching and Debugging the Kernel from Workbench*, p.99.

If the connection failed, go to *4.4.4 When a KGDB Connection Fails*, p.98.

4.4.4 When a KGDB Connection Fails

If the connection fails, try the following:

- Make sure the KGDB agent on the target is running. You can use the **lsmod** command on the target to see if the module is loaded.
- Make sure that the KGDB agent is using a functional network connection. For example, run the command **ifconfig** on the target, and double confirm that the kgdboe agent is using the right Ethernet connection.
- Double check that all the IP addresses for the target and host are correct, by running **ifconfig** on both the target and host.
- Try to ping the target from the host, and the host from the target, to test basic connectivity. The **ping** command uses TCP, and KGDB uses the more

restrictive unicast UDP packet, so you also need to check that you are on the same subnet, or have setup an “intranet to subnet” agent, as described earlier.

- Make sure that there are no firewalls on the host or target that may be blocking the connection.
- Make sure that there is only one copy of the Wind River WTX Target Registry running. You can try killing any running copy(s), and have Workbench start a fresh instance. You may need to restart Workbench.
- Make sure you have declared **127.0.0.1 localhost** within your **/etc/hosts** file, so that Remote Systems view can find the Target Registry on the host.
- You can turn on the debug log for the KGDB connection by adding the following to the advanced options box of the connection wizard.
  a. Right click on the connection.
  b. Select **Properties**.
  c. Fill in a location for the log in the **Backend Log**.
  d. Attempt the connect again.
  e. Select **Help > Collect Logfiles**, which will gather the connection log files.
  f. Forward this to the Wind River Support team.

4.5 Attaching and Debugging the Kernel from Workbench

This section describes how to access the target’s kernel and source code once a KGDB connection is made.

If the connection failed, go to *4.4.4 When a KGDB Connection Fails*, p.98.

Once the connection is established, Workbench shifts to the Embedded Debug perspective and the editor opens to the stopped location in the **kgdb.c** source file.

If you use the pre-built **vmlinux** file that comes with the pre-built kernel (in **wrlinux-1.x/boards**, the path to the source files will not be immediately known to Workbench. See step 3 on how to resolve this).

4.5.1 If the Compilation Unit Source Is Not Found

If the source window where **kgdb.c** should appear instead displays a message that says **Source not found for compilation unit**, your **vmlinux** file is either missing, or more likely does not point to the current location of your linux kernel source.

To fix this:

1. Click the **Browse** button in this window, and provide the path to your linux kernel's source directory.

   If you have performed a **make fs, make –C dist linux**, or a **make all**, the Linux source folder will have been expanded in your project's build folder, for example:

   /home/*user/workdir/myBoard***/build/linux-2.6.14-***type*

2. Click **Next** until you can click **Finish**.

   The location of **kgdb.c** and any other kernel source, will be resolved by Workbench.

You can view the mappings between the object files and the respective source files by right-clicking the entry in the Debug View window, and selecting **Edit Source Lookup**.

4.5.2 If the Target's Terminal Window Is Stopped

If you have a terminal window open to the target, you may see that it is stopped. For example, if you try to enter a command, you will find that you cannot.

To resume execution of the kernel, click **Resume** in the Debug view.

4.5.3 Setting and Removing a Sample Breakpoint

With the kernel running again, set a breakpoint as follows:

1. Select **Run > Add Expression Breakpoint**.
2. In the **Breakpoint Location and General Attributes** dialog, enter **do_fork** in the **Location Expression** box.
3. Click **OK**.

NOTE: If the **do_fork** location as a breakpoint does not work on this installation, note it and choose another kernel entry point. For possible explanations, see *4.2.5 KGDB and Safe Areas when Debugging the Kernel*, p.92.

4. Enter a command, for example **ls** (press **ENTER** only once,) on the target.

   It will not return because execution has stopped at the fork. The Source Editor should open to the **fork.c** source file. Clicking the Resume icon will allow the command to complete, and the next fork, for example another command, will again cause a break.

5. To complete this example, remove the breakpoint and click on the Resume icon to continue execution of the kernel.

6. To end the session, right-click the target connection in the Remote Systems view and select **Disconnect**.

4.6 Debugging User-Supplied Kernel Modules

Workbench includes a sample project to demonstrate kernel module debugging as discussed in this section. This example requires that you have access to the kernel source you used to build the kernel and that you have terminal or console window access to the target.

4.6.1 Building the Sample User-Supplied Kernel Module

To build the sample user-supplied kernel module:

1. Start Wind River Workbench running on your Linux host, if it is not already running.

NOTE: You must have the full kernel built in your project, or else the infrastructure for building kernel modules will not be in place.

2. Right-click the **kernel_build** build target, and select **Build Target**.

   The kernel will take about 10 to 20 minutes to build.

3. Select **File > New > Example** from the menu bar.

4. Select **Wind River Linux Kernel Module Sample Project**, then click **Next**.
5. Select **The Kernel Module Debug 2.6 Demonstration** and click **Finish**.

   A new project called **moduledebug_2_6** appears in the **Project Explorer**.
6. Select a platform project:
   a. Select **Project > Properties > Project References**.
   b. Check the box for **moduledebug_2_6**.
   c. Click **OK**.

      The **moduledebug_2_6** project disappears from the Project Explorer and appears inside the platform project.
7. Open the Platform project and find the sub-project **moduledebug_2_6**.
8. Right-click the sub-project **moduledebug_2_6**, then select **Build Project**.

   The module **moduleDebugExample.ko** appears in the directory of the **moduledebug_2_6** project. If you do not see it under the project icon, right-click the project and select **Refresh**.

   Remember: for the kernel module to build, the Linux kernel must be already built (by building the **kernel_build** build target in the platform project).
9. Right-click on the **moduledebug_2_6** sub-project and select **Properties > Build Properties**.

   You should see how the kernel root directory, target architecture, and cross compile prefix are set from the parent platform project.

You can add additional build options by appending them to the **Build command** entry **make**.

4.6.2 Debugging a Running User-Supplied Kernel Module

To debug a user-supplied kernel module that is running:

1. In the terminal view or console on the target, get the new file to the target, and use the **insmod** command to install the module.

```
kgdb_target_$ sftp your_host
kgdb_target_$ cd myhome/workspace/moduledebug_2_6
kgdb_target_$ get moduleDebugExample.ko
kgdb_target_$ quit
kgdb_target_$ insmod moduleDebugExample.ko
```

2. To make sure that you can see periodic output from this user module:

   a. Enter the following command:

   ```
   kgdb_target_$ klogd -c 8
   ```

   This command re-directs the kernel print statements to the console. You will see a message display every second or so. You may want to open an ssh or second Telnet session to remove this module when you are done.

   b. If the **klogd** is already running and there is no output, you can find the PID, kill the current instance, and restart the demon.

   ```
   kgdb_target_$ cat /var/run/klogd.pid
                 1084
   kgdb_target_$ kill 1084
   kgdb_target_$ klogd -c 8
   ```

   c. If you still do not see output, do the following:

   ```
   kgdb_target_$ cat /proc/kmsg
   ```

   Two messages should be internally printed: a “global” and a “local” message. Depending on the display level, you may see both messages interspersed.

   d. If the Workbench Remote Systems connection is connected to the target, disconnect. Once Workbench is again ready, reconnect to the target with the KGDB connection.

   e. If you have placed **moduleDebugExample.ko** in a location other than the root, adjust the target connection’s object mapping list to include that location, and then connect to the target. *Make sure that the host path has a forward slash at the end*.

3. Find the **moduleDebugExample.ko** entry in the Remote Systems view. If there is not a red “**S**” mark on its icon, right-click it and select **Load Symbols**.

   The red “**S**” should appear after a moment indicating that symbols were indeed loaded.

   If the symbols do not load, or if **moduleDebugExample.ko** does not appear in the Remote Systems view, go to the platform project’s **Object Path Mappings** screen, click **Add**, set the following values, and click **OK**:

   - **Target Path =** *(leave empty)*
   - **Host Path = /target/**

4. Right-click the Arch (OS) entry in the Remote Systems View and select **Attach to Core**. Click the **Resume** icon in the Debug view so that the kernel is Running.
5. Double-click **bpHere.c** in the moduledebug project. See that it appears in the central editor view.
6. Right click in the left-hand gutter of **bpHere.c** at line 21 **(printfk(KERN_ALERT …)** of the routine **putBreakPointHere**. Select a **System Context** break point.

NOTE: If you find that the break point cannot be planted, make sure you added the Object Mapping.

7. Once it breaks (as seen in the Debug view), try **single steps** in and then out of this procedure. Then click **Resume**.
8. When it breaks again, disable the break point and click **Resume**.

You can remove the module from the kernel with the **rmmod** command:

```
kgdb_target_$ rmmod moduleDebugExample.ko
```

4.6.3 Stepping into User-Supplied Kernel Module init from module.c

The section describes how you can place a breakpoint in the kernel's module load routine, and step directly into a loading module's **init** routine.

This technique can be used in many ways beyond this specific example. You can also see the next section for fast ways to break within specific kernel module **init** routines, loaded or not.

1. Select **File > Open File** and browse to the **kernel** subdirectory of your build's Linux-2.6.14 source tree root.

   For example, **module.c** might be found at:

   */home/user/workdir/myboard***/build/linux-2.6.14-***type***/kernel/module.c**
2. Double-click the **module.c** file to open it in the editor.
3. Go to the **sys_init_module** routine and find the lines below that call the module's init procedure (press **CTRL-F** and search for the string "Start the Module").

```
    /* Start the Module */
=>  if (mod->init != NULL)
    ret = mod->init();
```

4. Right-click in the gutter to insert a **System Context** breakpoint there.

NOTE: If the break point cannot be planted, make sure you added the Object Mapping as described earlier. Also, do not single step from the break at **sys_init_module** to this location – you will hit spin locks that will block your KGDB connection

5. On the target, start the module:

```
kgdb_target_$ insmod moduleDebugExample.ko
```

   In a moment, the Debug view shows the System Context as stopped and **load_module** highlighted; the editor opens **module.c** file to the location of the breakpoint. Note that the target window is hung at the **insmod** command.

6. Ensure that the **moduleDebugExample.ko** is registered by the Remote Systems view. Examine the window for your target connection, and look for **moduleDebugExample.ko** in the list on installed kernel modules, under **Processes**. If is not present, take the following steps to synchronize the remote systems view:

   a. Un-expand the icon for the target labeled **Processes**.

   b. Click the **Refresh** button in the Remote Systems view's tool bar.

   c. Re-expand the icon for the target labeled **Processes**.

   d. It the module does not yet appear, repeat the above steps.

7. Ensure that the symbols for **moduleDebugExample.ko** are loaded. Examine the Remote Systems view for your target, and look for **moduleDebugExample.ko** in the list on installed kernel modules, under **Processes**.

   The icon for this module should be decorated with a red **S**. If it is not, right click the module and select **Load Symbols**.

8. You can now single step into the initialization routine of the kernel module.

9. To resume processing, remove the breakpoint and click the **Resume** icon in the Debug view. The **insmod** command completes.

10. You can remove the module from the kernel with the **rmmod** command:

```
kgdb_target_$ rmmod moduleDebugExample.ko
```

11. When you remove the module, notice that **moduleDebugExample.ko** is no longer registered in the Remote Systems view.

4.6.4 Placing Breakpoints in Modules Not Yet Loaded

When a kernel module is loaded, it tries to load the symbol table and place any pending breakpoints.

To use this as a fast way to break in the **init** routine of a kernel module:

1. Right-click the GDB target connection entry in the Remote Systems view.
2. Select **Properties > Object Path Mapping**.
3. Click **Add.**
4. Leave the **Target Path** empty.
5. Click **Browse** for the Host Path, and select **workspacedir/moduledebug_2_6** for the directory where the generated **moduleDebugExample.ko** file is found.
6. Click **OK** for the path, and **OK** for the connection's properties dialog.
7. Make the connection to the target.

   If that connection is already active, disconnect and reconnect to assert the new mappings. Click **Resume** so that the target is running.
8. Right-click in the left gutter of the **bpHere.c or modules.c** source file at the location indicated for the breakpoint, and select **Breakpoint > Add Breakpoint (Scope = "System Context")**. You can change the scope of any breakpoint by clicking it in the Breakpoints view.

NOTE: If you see a dialog stating that the location for the breakpoint is not available, remember that the module is not currently loaded. Click **OK** to continue. If you close the source file you will be able to see that Workbench opens it when the breakpoint is hit.

9. Load the module on the target.

   ```
   $ insmod moduleDebugExample.ko
   ```

   Observe the breakpoint being hit, and the source being opened to that location.
10. You can now debug the **init** routine, or resume the execution.
11. When you are finished, remove the module:

    ```
    $ rmmod moduleDebugExample.ko
    ```

    If you may see another dialog stating that the location for the breakpoint is not available, this is all right because the module is unloaded. Click **OK** to continue.

4.6.5 Providing Object Path Mappings to All Modules

For Workbench to find the appropriate source file for a module, it requires the path to the object files. Unfortunately, the kernel modules for a kernel may be scattered across many subdirectories, making it a tedious task to specifying all of them.

To easily gather all those paths, and make them available to Workbench.

1. In Workbench, build the **kernel_build** target so that all kernel files are available.
2. Select **Project > Open Shell** and then:

   ```
   $ cd build/linux-version-type
   ```

   (The shell now opens automatically in the *project*_**prj** directory.)
3. Create a link to all kernel modules within the Linux kernel, and place them in this directory:

   ```
   $ foreach file (` find . -name "*.ko" -print`)
   foreach? ln -s $file
   foreach? end
   $
   ```
4. Add this object path mapping to your KGDB connection:
   a. Right-click the specific KGDB target connection entry.
   b. Select **Properties > Object Path Mapping**.
   c. Click **Add**.
   d. Leave the **Target Path** empty.
   e. **Browse** for Host Path; select *projectname*_**prj/build/linux-version-***type*.
   f. Click **OK** for the path, then **OK** for the target connection's properties dialog.
5. Make the connection to the target. If that connection is already active, disconnect and reconnect to assert the new mappings.

You can now place breakpoints in any kernel module source, have it planted by Workbench, hit those breakpoints, and have Workbench always be able to bring up the correct source file.

4.6.6 Using the testmod Enhanced Sample Kernel Module

This sample kernel module demonstrates several features:

- A simple kernel module. The file **main.c** initializes and removes the module.
- Parameter passing.
- A kernel timer. The file **timer.c** manages the timers.
- Using **sysctl** to change the timeout. The file **settings.c** manages the **sysctl** bindings.

1. Create a new kernel module project:
   a. Select **File > Example** from the menu bar.
   b. Select **Wind River Linux Sample Project**, and click **Next**.
   c. Select **Wind River Linux Kernel Module Sample Project**, and click **Next**.
   d. Select the **Example Kernel Module Demonstration** and click **Finish**.

      A new project called **testmod** appears in the **Project Explorer**.
2. Select and open an existing platform project in Workbench:
   a. Select **Project > Property > Project References**, then select **testmod**.

      Note that the **testmod** project disappears from the Project Explorer and appears inside the platform project.
   b. Open the platform project, and find the sub-project **testmod**.
   c. Right-click **testmod**, and select **Build Project**.

      The module **testmod.ko** appears in the directory of the **testmod** sub-project. If you do not see it under the project icon, right-click on the project and select **Refresh**.

NOTE: For the kernel module to build, the Linux package must be already built (by building the **linux** build target for example).

3. In the terminal view or console on the target, get the new file to the target, and use the **insmod** command to install the module.

```
kgdb_target_$ ftp your host
kgdb_target_$ cd myhome/workspace/moduleDebug
kgdb_target_$ get testmod.ko
kgdb_target_$ quit

kgdb_target_$ insmod testmod.ko timeout=2340
```

4. Verify that the console output shows **testmod**:

```
kgdb_target_$ dmesg | tail
Loading testmod
timeout is 2340
```

5. Use various additional **sysctl** supported commands to work with this module:
   - **sysctl -a** shows all variables.
   - **sysctl dev.testmod.timeout** queries variables.
   - **sysctl dev.testmod.timeout=5000** sets variables.

   You can also query and set variables using **/proc/sys/dev/testmod/timeout**.
6. When you are finished, remove the module with:

```
kgdb_target_$ rmmod testmod.ko
```

4.6.7 Creating a Custom Kernel Module

To use Workbench to create a custom kernel module project.

1. Select **File > New > Example > WR Linux Kernel Module Project**.
2. Give the project a name, for example **custom_klm**. Click **Next**.

   To attach this project to a platform project within Workbench, leave these values empty. To connect this custom kernel module to a platform project outside of Workbench:

   a. Fill in that **Linux Kernel Directory**, its **ARCH**, and its **CROSS_COMPILE** value. (You can always edit these values later via the project's Build Properties.)

   b. Click **Finish** when done.
3. Right-click the project, and select **Import > General > File System**.

NOTE: We are going to borrow the code for this kernel module from an existing Wind River Linux sample project. You can import your own module code if you have some available.

4. Browse to *installDir*/**wrlinux/***version*/**samples/moduledebug_2_6**.
5. Select the files **bpHere.c** and **moduledebug.c**, and click **Finish**.

6. You can now build this project:
   - If you have attached this project to an external platform project.
   - If you have attached this to an existing Workbench platform project as described in *4.6 Debugging User-Supplied Kernel Modules*, p.101.

This module is now ready for deploying and testing.

4.7 Using QEMU for KGDB Kernel Debugging

QEMU offers two types of kernel-mode debugging, using KGDB:

- ICE-like debugging using the KGDB agent built into QEMU. This allows you to debug areas such as exception vectors and Ethernet drivers, that regular KGDB-on-target debugging normally does not allow. (See *4.2.5 KGDB and Safe Areas when Debugging the Kernel*, p.92.)
- KGDB over Ethernet, that is, a kgdboe agent running inside your kernel, on your target, loaded via the **modprob** command. This type of connection can show you, for example, the context that has caused a **panic( )** state in the target.

This section explains the ICE-like connection debugging.

To create an ICE-like connection and open the target kernel in the debugger:

1. Click the **Create a New Target Connection** button in the Remote Systems view toolbar.
2. Select **Wind River Linux KGDB Connection** and click **Next**.
3. Select **Linux KGDB via Ethernet** and click **Next**.
4. In the Linux KGDB Connection Properties dialog, choose or enter the following, then click **Next**.

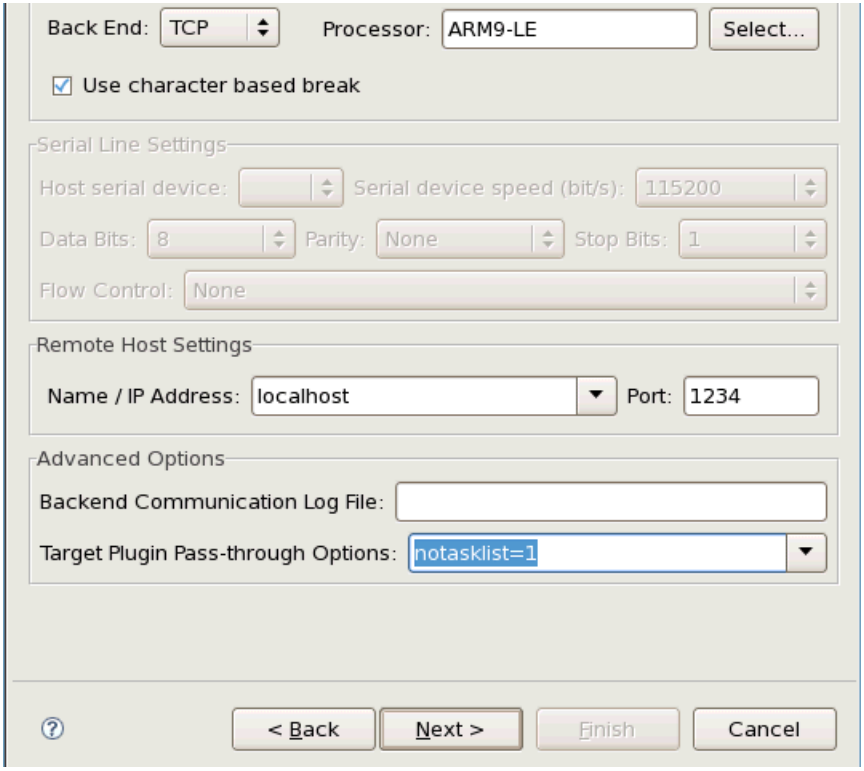
   Back End: **TCP**

   CPU: **ARM9-LE** (or any supported QEMU connection)

   Name/IP Address: **localhost**

   Port: **1234**

   Target Plugin Pass-through Options: **notasklist=1**

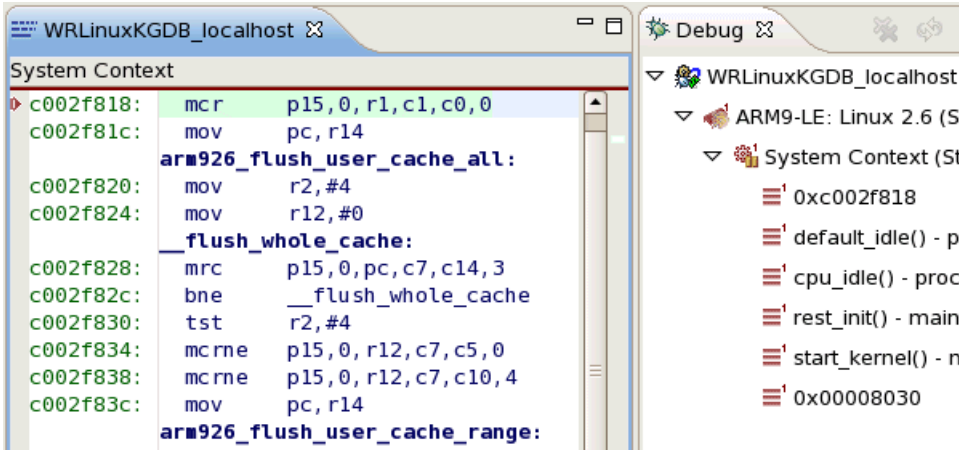


5. In the **Kernel image** field in the resulting **Target Operating Systems Setting** dialog box, enter the name and path of the Linux kernel symbol file: *project*_**prj/export/***boardName*-**vmlinux-symbols-***version_fsType*, for example:

*project*_**prj/export/arm_versatile_926ejs-vmlinux-symbols-WR2.0ao_standard**

6. Click **Next** as needed to reach the Connection Summary dialog.
7. Enter a name for the new connection.
8. Check **Immediately connect to target if possible**.
9. Click **Finish**.

   The new connection appears in the Remote Systems view, and the debugger opens onto the kernel running on the emulated target, as in the following figure:



At this point, the debugger has attached the KGDB agent in QEMU on the target side, halted the target, and stopped in the kernel.

10. In the Debug view, press the **Resume** button to resume the target operation.

Now, you can do any kernel mode debugging that you would normally do using ICE with a real target.

4.8 Enabling and Disabling KGDB in the Kernel

This section contains notes about enabling and disabling KGDB in the kernel.

By default, KGDB is enabled in the pre-built and generated Wind River Linux kernels. Here are the steps to disable KGDB, used for example when transitioning to production builds:

1. Double-click the project's Kernel Configurator icon in the Project Explorer.

   If you have not previously extracted the kernel, answer **Yes** when Workbench prompts to extract the kernel files.

2. Open the Kernel Hacking tree.
3. Double-click the **Compile the kernel with debug info** menu item to toggle between **Yes** and **No**.
4. Select **File > Save**, to capture the configure changes.
5. Right-click the project's build target **kernel_rebuild**.

The new kernel and **vmlinux** file will be available in the workspace's *projectname*_**prj/export** directory.

5 Configuring Wind River Linux Platforms

5.1 Introduction 115

5.2 Configuring Wind River Linux Platform Kernels 116

5.3 Adding Kernel Modules to the Platform 117

5.4 Adding Build Targets for Analysis Tools 120

5.5 Working with Packages and Target File Systems 121

5.6 Using Quilt to Manage Patches 133

5.7 Automating Target Deployment 138

5.1 Introduction

This chapter explains how to configure platform kernels; add kernel modules to a platform; work with packages, build targets, and target file systems; handle RPMs; manage patches; and automate target deployment.

5.2 Configuring Wind River Linux Platform Kernels

When you build a kernel in a Wind River Linux Platform project, you can easily view and modify the current kernel configuration with Workbench. This section summarizes the major uses of the kernel configuration tools. For detailed reference information on the **Kernel Configuration** node, refer to *Wind River Workbench User Interface Reference: Kernel Configuration Editor.*

Workbench provides a **Kernel Configuration** node for Wind River Linux Platform projects that presents the standard Linux configuration choices in a convenient GUI that quickly shows inter-dependencies and provides additional information on each configuration option. If you prefer, you can still use the **menuconfig** or **xconfig** interface, either from the command line or by clicking the **menuconfig** or **xconfig** nodes in the platform project.

5.2.1 Kernel Configuration Node

When you double-click **Kernel Configuration** in the project folder, a tab with the name of your project opens in the central view, displaying a configuration window with two tabs: **Overview** and **Configuration**.

Overview Tab

The **Overview** tab displays the project configuration. This corresponds to the choices you made when creating a new Wind River Linux Platform project. The command-line arguments supplied to the **configure** command to create the initial configuration are shown at the bottom of the view.

Configuration Tab

The **Configuration** tab consists of two windows. The top window presents the items for configuration that may be familiar to you from using **menuconfig** or **xconfig** to configure kernels. The lower window consists of three tabs, **Help**, **Dependencies**, and **Kconfig**, that provide additional information on the items you select in the top window.

NOTE: By moving your cursor around inside the Kernel Configuration view, you can find the drag points to enlarge view sections.

When you select an item in the top window, Help appears in the **Help** tab. Select the **Dependencies** tab to see any items your selected item depends on, or items that depend on it. Double-click any item dependency to move to it in the top

window. Select the **Kconfig** tab to see the definition of the item and the file system location where it is defined. If you click the file path link shown in blue the file is added to the Edit view and you are placed at the location of the item definition.

Create a new kernel configuration by modifying the settings of menu items in the top window. The current configuration status of items is indicated by the icons associated with them. (Refer to *Wind River Workbench User Interface Reference: Kernel Configuration Editor* for a detailed description of the icons in the menu.)

The Configuration view does not show all items as some of them are disabled due to dependency conditions. To view all items, including disabled items, right-click in the Configuration view and select **Filters > Show All**. The disabled items are shown as greyed-out. You can still select them and view the **Dependencies** tab to understand the reasons they are disabled.

Modifying the Configuration

For example, if you want your kernel to include support for PCMCIA or CardBus cards, find **Bus options** in the scroll-down list, click the arrow to expand it and then click the arrow next to **PCCard**. This displays the entry for **PCCard support**. This is a tri-state configuration item which can be set to **n** for not included, **m** for included as module, or **y** for built-in. Double-click the item to change the setting. See the **Help** tab for details on what the settings mean for the particular item.

To find options, for example those concerned with USB support, select **Edit > Find** and enter the string **USB**.

To generate the new kernel, right-click **reconfig** in the project folder and select **Build Target**.

To save the configuration file, select **File > Save**.

5.3 Adding Kernel Modules to the Platform

You can create a kernel module as a subproject of a kernel project, and the kernel module subproject will inherit the kernel build properties. This makes creating a kernel module a simple task.

Workbench comes with a sample kernel module project that you can add to a kernel project. See *4.6 Debugging User-Supplied Kernel Modules*, p.101.

5.3.1 Creating a Custom Kernel Module

You can configure the build system for a custom kernel module created from your source files once you have created a platform project. In this example, you will directly attach the new module to an existing platform project. Alternatively, you could skip that step and enter the **KERNEL**, **ARCH**, and **CROSS_COMPILE** values directly for an external kernel, or even leave these fields blank and attach or set them at a later date.

When you build this custom kernel module, note that the generated Makefile automatically includes the module's source files, and the module name has been set from the project name.

1. Right-click in the Project Explorer and select **New > Wind River Linux Kernel Module Project**.
2. Assign a name to your kernel module project and click **Next**.
3. Select a platform project as a superproject for this kernel module and click **Next**.
4. Note that the **Linux kernel information** at the bottom of the dialog is filled-in because the values have been imported from the platform superproject.
5. Click **Finish**.

   The kernel module project now appears as subproject of the platform superproject in the Project Explorer.
6. Import source files:
   a. Right-click the new kernel module project and select **Import > General > File System**.
   b. Browse to *installDir*/**wrlinux-2.0/samples/moduledebug_2_6**.
   c. Click **OK**.
   d. Select the files **bpHere.c** and **moduledebug.c**.

      You could of course import your own source files. The files in this example demonstrate kernel module builds and testing: **moduledebug.c** provides the module init and exit points, and **bpHere.c** sets a breakpoint.
   e. Click **Finish.**

You can now build the module, deploy it, and test it.

5.3.2 Moving Kernel Module Projects

To move the kernel module project from one platform project to another, use the following procedure:

1. Right-click the kernel module project and select **Build Clean**.
2. Right-click the platform project that currently contains the kernel module subproject and select **Properties**.
3. Click **Project References**, deselect the kernel module project, and click **OK**.

   The kernel module project reappears in the Project Explorer at the project level.
4. Right-click the platform project that you want to contain the kernel module subproject and select **Properties**.
5. Click **Project References**, select the kernel module project, and click **OK**.

   The kernel module project appears as a subproject of the new platform project.
6. Right-click the kernel module project and select **Build Clean.**

5.4 Adding Build Targets for Analysis Tools

You can add build targets to your platform project in the Project Explorer for any rules you add to **Makefile.wr**.

If you are going to use Run-Time Analysis Tools, it can be useful to create a build target to make both Run-Time Analysis Tools and the filesystem. The build of Run-Time Analysis Tools copies the correct target agents to the filesystem and also builds a KAL module that matches the kernel. The build of the filesystem then includes the necessary Run-Time Analysis Tools files.

Note that Memory Analyzer and Performance Profiler require that the KAL module must match the kernel, so if you rebuild the kernel you should rebuild Run-Time Analysis Tools and the filesystem, as provided by the build target you add in this section. Alternatively you could build only Run-Time Analysis Tools and then copy the **KAL.ko** module to the target.

Creating the Run-Time Analysis Tools Build Target

To add a build target that makes Run-Time Analysis Tools and then makes the filesystem, do the following:

1. Double-click **Makefile.wr** in the platform project to open the makefile in the Editor.
2. Add the following rule:

```
Run-Time Analysis Tools :
    @make -C $(DIST_DIR) Run-Time Analysis Tools ;
    @make fs
```

3. Save **Makefile.wr**.
4. Right-click the project and select **New > Build Target**.
5. Enter the name **Run-Time Analysis Tools** and click **Finish**.

   A new **Run-Time Analysis Tools** build target appears in your platform project.

To use the new build target, right-click the **Run-Time Analysis Tools** build target and select **Build Target**. Workbench builds Run-Time Analysis Tools and then the filesystem.

5.5 Working with Packages and Target File Systems

Wind River Linux Platform projects have a **User Space Configuration** node that allows you to:

- Explore the target file system, view the contributing packages, and delete or add files to customize and fine tune the file system content.
- Work with packages (RPMs) involving the target file system. You can add, remove, import, build, and debug packages.

5.5.1 Introducing the User Space Configuration Node

The User Space Configuration node in the Project Explorer lets you handle packages as well as explore the target's file system.

To familiarize yourself with this node:

1. Double-click **User Space Configuration** in the Project Explorer.

   A window opens in the central view.

2. Click the **Filesystem** tab in the lower-left of the new window.

   The **Filesystem Layout** window opens. See *5.5.2 Exploring the Target File System*, p.121.

3. Click the **Packages** tab in the lower-left of the new window.

   The **Packages** window opens. See *5.5.3 Handling Target File System Packages (RPMs)*, p.123.

5.5.2 Exploring the Target File System

This section explains how to work with the Filesystem Layout window that opens when you click the **Filesystem** tab in the lower corner of the Package Configuration window that opens from **User Space Configuration**.

The Filesystem Layout window lets you manage the file system in the **export/dist** directory before Workbench creates the **dist.tar.bz2** image. This image is expanded into an NFS mount directory, whereas you cannot NFS mount the **export/dist** directory. This image is also used by QEMU emulation (see *5.6 Using Quilt to Manage Patches*, p.133).

The **export/dist** directory is *the* file system image for all configured projects. Regular targets take that image, then create the **dist.bz2 tar** image as the last step in creating the file system. QEMU can use that image directly, saving a step in expanding the file system into some location. QEMU also can use the "fakeroot" information behind the **export/dist** image, to describe which files "belong" to root and such for the target image (and appear that way in the live target file system), as opposed to those belonging to the host's root account.

The Filesystem Layout window displays:

- A list of directories and files that comprise the file system of the target that you are building. To see this mode, click the **View** icon to the far right of the **Pattern Filter** field, and choose **Flat**.
- A list of packages within the file system of the target that you are building. To see this mode, click the **View** icon to the far right of the **Pattern Filter** field, and choose **Packages**.

Filesystem Layout components act as follows:

- The **Pattern Filter** field lets you enter and apply filters to the view, allowing you to specify, for example, all files with a **.pl** extension.
- The **Filter** icon to its right activates the filter, letting you mark the results for deletion if you wished. The filter works on all the packages, so you could remove unneeded Perl scripts from them all, for example.
- The **View** icon to the far right selects the file system or package view. The file and package list specify file, mode, size, type, owner/group, package, and location. You can change file and package attributes, such as read/write permissions and ownership. Right-click to mark directories to be created in or removed from the file system; you can also create nodes, pipes, and devices in the generated file system.
- The **Filesystem** line below the list annotates the size of files or packages added or removed, along with the total size of the file system currently available.
- The **Log** tab reports on the number of packages, files, and so on.

5.5.3 Handling Target File System Packages (RPMs)

This section explains how to work with packages (RPMs) involving the target file system. You can add, remove, import, build, and debug packages.

NOTE: RPMs are not supported in small file systems such as glibc_small and uclibc_small, because too many packages would be required and the file systems could no longer be considered small.

The Packages window displays a list of installed packages in the pane below the heading **Installed Packages**. If you look in the **pkglist** file in the *project*_**prj** directory, you will see the same list of files. In Workbench, each package is listed with its version number and the amount of space it takes.

You can move the cursor around the edges of the Package Configuration view, and below the buttons, to find the drag points that enlarge view sections.

To get more information on any package, select it and look at the tabs at the bottom of the view:

- The **General** tab gives a brief description of the package.
- The **Advanced** tab gives an editable URL to the RPM for the package, and states the architecture.
- The **Contents** tab lists the various files and directories that comprise the package.
- The **Dependencies** tab displays package interdependencies in two windows:
  - The **Requires** window lists the packages that are required by the package you have selected.
  - The **Required By** window lists the packages that require the package you have selected.
- The **Options** tab lets you turn debugging on or off for particular packages.
- The **Targets** tab presents 10 buttons that let you choose build targets for the package: build, rebuild, prepatch, postpatch, patch,compile, install, custom, clean, and distclean.
- The **Log** tab primarily lists the additional RPMs that will be generated by the installed packages.

Working with the Controls for Packages

The Packages window provides buttons and context menu entries to help you work with one or more packages.

Use the package buttons as follows (from left to right):

- **Remove** takes packages out of the **pkglist** file and puts them into the Installable Packages list.
- **Check** confirms the integrity of the package list by testing all package dependencies, as registered in the respective and collective rpm file set. The integrity cannot be confirmed entirely because of potential conflicts with the original dependency content from upstream and in the respective customer package data.
- **Add** moves packages from the Installable Packages list into the **pkglist**.
- **Import** adds packages that are not in the Installable Packages window.

Use these additional context menu entries (listed alphabetically) as follows:

- **Erase** removes the respective package's RPM file from the live target (**Erase** is the official rpm command).
- **Force Install** installs the respective package's RPM file on the live target, independent of the architecture or package dependencies.

NOTE: This runs *installDir***/wrlinux-2.0/scripts/launch_script.sh**, executing the following command at line 62: launch_execute rpm_force $RPM -iv --ignorearch --nodeps --force $* (see **man rpm** to tailor this as needed).

- **Install** causes the respective package's RPM file to be installed on the live target.
- **Open** places the actual **pkglist** file in the editor to provide direct text editing of the actual file. Use the **Reload** command to see any changes.
- **Reload** rescans the **pkglist** file, and reloads the package lists in this view.
- **Replace** does an rpm **Erase** and and an **Install** on the target for the respective RPM file.
- **Revert** returns the **pkglist** file's content to the original content (as it was when this view was opened).
- **Save** adopts any changes.

Force Installing Packages

The **Force Install** context menu entry lets you install an RPM file without regard to any mismatch between the host and target architectures, or package dependencies that cannot be fulfilled.

Because the dependencies as defined by **rpm** are rigid, they may not apply to your platform. For example, suppose you have removed perl or python (using perhaps the Workbench File System Layout tool: see *5.5.4 Using the File System Layout Configuration Tool*, p.129). Now, suppose you want to install one of the 150 or so packages that happen to include perl or python scripts. Those packages will fail to install because they have a rigid dependency on perl or python.

If this is a dependency that does not concern you, you can use **Force Install** to ignore it. Also, you can use **Force Install** to get around dependencies that are not quite correct, allowing you to avoid debugging and fixing packages.

Example use of **Force Install**:

1. Erase (remove) **iptables-ipv6** and then **iptables**, which it requires:
   a. First, choose **iptables-ipv6**, because of the dependency.
   b. Select **Erase** from the context menu.
   c. Choose **iptables**, then **Erase**.

   Notice that you cannot install **iptables** now, whether you use **Install** or **Force Install**, because **iptables-ipv6** requires it but is not itself installed.
2. Verify that line 62 of *installDir*/**wrlinux-2.0/scripts/launch_script.sh** reads: launch_execute rpm_force $RPM -iv --ignorearch --nodeps --force $*
3. Now, use **Force Install** to install **iptables**. It should succeed.

Removing Packages

For example, you could do the following to remove packages:

1. Select the **Dependencies** tab.
2. Select a package such as **openssl** from the **Installed Packages** list that contains entries in both the **Required** and **Required By** windows.
3. Click **Remove** to remove the installed package that you have selected.

A dialog box appears that lists all the packages that will be removed (the selected package and those packages it is required by) and the total space taken by the packages.

4. Click **OK** to remove the packages.

The removed packages now appear in the **Installable Packages** list. Select F**ile > Save** and look in the **pkglist** file. The packages will have been removed from the list in that file.

Installing Packages

To reinstall the same packages, do the following:

1. Select them from the **Installable Packages** list and click **Add**.

   A dialog box prompts you with the package(s) to be added and the total space involved. If other packages are required for the package you have selected to be added, they are shown in the dialog box as well.

2. Click **OK**.

   The package(s) appear in the **Installed Packages** list.

You can add back the other packages that you removed, then select **File > Save** to restore the package configuration to its original condition. If you look in the **pkglist** file, all packages are now listed there.

Note that you can select multiple packages to add or remove at one time.

Installing Multi-lib Packages and Exporting Their Sysroots

To be able to use the Package Manager to set multi-lib RPMs in the **pkglist** file:

1. Create a Cavium platform project.
2. Add the following two lines to **pkglist**:

```
glibc.mips64_octeon
libgcc.mips64_octeon
```

3. Export the sysroot. It will then include these extra multi-libs.
4. Add this sysroot to **Window > Preferences > WR > WR Linux > Sysroots**.

You can now build **mthread**, for example, for both the 32-bit **n32_octeon** and the 64-bit **octeon** build specs.

NOTE: If you see a pointer size warning for the **octeon** spec, it does not affect the build success.

Importing Packages

The **Import** button lets you import (install) packages that are not listed in the **Installable Packages** window. Use this feature to set up the imported package in the build system.

To import packages:

1. Click the **Import** button.

   The Package Import Tool appears.

2. Click the package type: **File**, **Wget**, or **WR Linux Package**.

3. Browse for the package to import.

   When you have selected the package, the package list, package name, and version appear in the Package Import Tool screen.

4. Click **Go**.

   A shell window opens to display the import activity.

5. Press **RETURN** as needed to return to the import tool from the shell.

6. Examine the progress window to see that it has completed satisfactorily.

7. Press **Close**.

   The import concludes.

You still may need to complete the package, for example, to change it for cross compilation.

Building Packages

Select the **Targets** tab for a set of buttons of build targets for the selected package.

The build targets are the usual with the addition of **prepatch**, which allows you to prepare the package for patching (see *5.6 Using Quilt to Manage Patches*, p.133).

The **prepatch** button:

- Extracts any RPMs for the package that you selected.

- Registers kernel patches.
- Registers patch files.
- Prepares the Quilt patches directory (see *5.6 Using Quilt to Manage Patches*, p.133).

Debugging Packages

The debugging features available via the User Space Configuration node give you the ability to select individual packages and build them with debugging turned on or off, regardless of the rest of the pre-built packages in your project. The result is a very efficient use of the debugger and development/debug time.

To enable and disable debugging for a package, select the package, the **Options** tab, and select or clear the **Debug** box.

To enable debugging for a package:

1. Select the package.
2. Select the **Options** tab.
3. Check the **Debug** check box.
4. Click **Apply**.

If you look in the **pkglist** file and find the package you just marked for debugging, the package name has **BUILD_TYPE=debug** next to it.

To turn off debugging:

1. Select the package.
2. Select the **Options** tab.
3. Uncheck the **Debug** check box.
4. Click **Apply**.

5.5.4 Using the File System Layout Configuration Tool

The File System Layout Configuration Tool manages target file system contents before they are finalized in the **export/dist** directory and the **dist.tar.bz2** file.

This tool consists of:

- A script that controls the file system contents
- A Workbench counterpart to graphically manage it

See the *Wind River Linux Platform User's Guide* for more about the script itself.

The following subsections explain the context menu entries for the file system configuration tool, as well as how to use the tool to:

- Examine meta properties of the files in the file system.
- Add and remove files and directories to and from the file system.
- Work with packages.
- Add devices and change ownership.

Using the Context Menu Entries

The Filesystem Layout Configuration Tool provides the following context menu enries:

- **Open** brings up the selected file resource in an editor, if the file exists in the local file system (for example, if the file is an added file). It is not enabled for files that are located in an rpm archive, except for those files located in *project*/**filesystem/fs** and in an rpm named *board*-**config.rpm**.
- **New**
  - **Folder** creates a new folder below the selected resource.
  - **Link** creates a new symbolic link. You must provide the link location and target. The location specifies where in the file system the link should be placed. The link value specifies the target to which the link should point (this value is automatically populated with the current selected resource). For example, consider the command line usage of **ln**:

    ```
    ln -s /tmp /foo
    ```

    The link value is **/tmp**, and the link location is **/foo**.

  - **Device** creates a new character or block device below the selected resource.
  - **Pipe** creates a new pipe below the selected resource.
- **Add** adds a single file from the local file system below the selected resource.
- **Add Folder** adds a folder recursively from the local file system below the selected resource.
- **Remove** deletes resources recursively below the selected resource. If the resource is located in an rpm archive, it is marked with an "-" overlay icon. If the resource is a previously added local resource it is removed from the tree.
- **Properties** changes the properties of the selected resource(s), allowing you to change the umask, the owner, and group properties.
- **Revert to saved** undoes all changes since the last save.
- **Revert resource** undoes the change stored in the changelist for a single resource (only if a changelist is available).

  A *changelist* is a locally stored file (*project*/**filesystem/changelist.xml**) that contains a list of changes to be used by the **make fs** command.

NOTE: Revert resource only reverts the selected resource. It does not perform a recursive revert. You can use **Revert all** to revert all files and directories, or you can manually select each resource one at a time and use **Revert resource**.

- **Revert all** undoes all changes stored in the changelist (only if a changelist is available).
- **Reload** reloads the changelist from the file system.
- **Save** saves the changes to the **changelist.xml** file.
- **Import** imports a changelist file and merges it with the current changes.
- **Export** exports the current changes to a specific location and file.
- **Filter**
  - **Resources** filters resources based on whether they have been added or removed, or have had a change in status, depending on whether the resources are files, folders, links, and so on.
  - **Packages** filters based on the list of rpm archives and allows including and excluding packages from the view.
  - **View** switches between the flat and the package-based view.

- **Expand** opens all levels recursivly below the selected resource.
- **Collapse** closes all levels recursively below the selected resource.

Examining Meta Properties of the Files in the File System

To examine file properties:

1. Create a platform project.
2. Double-click the project's **User Space Configuration** node.
3. Click the Filesystem tab in the lower left, and observe the file system tree.
4. To examine the meta properties of files:
   a. Open the **bin** directory, and examine the first file, for example, **arch**.
   b. Observe the following:
   - The mode string (-rwxr-xr-x)
   - The file's size (4230)
   - The file's type, for example **ELF 64-bit LSB executable** for a common_pc_64 project
   - The owner and group
   - The file's parent package (**util-linux-2.13-0.52_WR_a**)

Adding Files and Directories to the File System

To add files and directories:

1. Right-click on the directory /**usr** and select **New > Folder**.
2. Enter the name **samples**.

   Observe that the new folder is added, with a "+" icon.
3. Right-click the directory **/usr/samples** and select **Add**.
4. Browse to the workspace, and select a build sample project binary, for example, **/tmp/workspace/mthread/common_pc-glibc_std-i686/mthread/Debug/mthread.out**.

A new file appears with a "+" icon, and the **Location** column contains the host file system source path.

NOTE: This step assumes that you have built an application project, specifically, for this example, the mthread application.

Working with Packages

To view parent packages and remove packages:

1. Right-click and select **View > Packages**.

   The tree shows each package, and its part in the target tree.

To remove a package, for example, the "optional" Perl package, and all Perl scripts:

1. Open the tree to **perl-5.8.7-1_WR_a/usr/bin**.
2. Right-click this directory, and select Remove.

   All files in this directory now have a "-" icon.
3. Right-click **perl-5.8.7-1_WR_a/usr** directory, and select Remove.

   Now all resources under **Perl** are marked with "-" icon
4. Set the **Pattern Filter** value to ***.pl**.

   The **/usr** directory that appears already has all contents marked with "-" icon.
5. Reset **Pattern Filter** to blank, and click **Return** in the text box to restore the default tree filter.

Adding a Device and Changing Ownership

To add a device:

1. Right-click the **/dev** directory, and select **New > Device**.
2. Use the settings **Char,"luckyDevice",7,11**.

   The new device appears, with the package name of **added**.
3. Right-click the device, and change the owner to **lady**, and the group to **luck**.
4. Right-click this new device, and select **Remove**.

5.6 Using Quilt to Manage Patches

The Wind River Linux build system works with two patching models.

- The previous model, called the *Classic patching model*, which carries forward from wrlinux-1.3 and is based on patch list files and the program **patch**.

NOTE: Wind River strongly encourages you to use the Quilt patching model.

- The new default model, called the *Quilt patching model*, which is based on **patch**, but provides an additional, rich set of patching features, such as pushing new patches to go forward, popping patches to go backward, capturing development changes into new patches, deriving patch annotation data, and more.

The Quilt patching model is the default for both command-line- and Workbench-configured projects as it provides many advanced patching features.

NOTE: The **-enable-quilt=[no | yes]** configure option overrides the default patching model (Quilt) for project configuration.

Also, the Configure Options screen defaults to **Enable Quilt** for new projects..

For recommended reading, see Andreas Grünbacher's "How To Survive With Many Patches or Introduction to Quilt," **http://www.suse.de/~agruen/quilt.pdf**.

5.6.1 Setting Up Quilt to Manage Patches

The following procedure exemplifies how to apply patches to a project, rebuild the project, and recognize the **patches** node within the rebuilt project:

1. Create a Wind River Linux platform project.

   Choose board, rootfs, or kernel as you like.

2. In the Project Explorer, right-click the platform project and choose **Add Link to Build Folder**.

3. Double-click **User Space Configuration** under the platform project folder.

   Reply **Yes** if asked if you want to initialize the prebuild PM package folder.

4. Select a package on the left list, for example, **boa** or **busybox**.

   The list of packages varies depending on which rootfs you have chosen.

5. Under the Targets tab, click the **patch** button and wait until ready.

   The user package is extracted to your workspace and Workbench applies all patches.

6. Right-click the **build** folder below the project folder (in the Project Explorer) and choose **Refresh**.

   A *package*-*version* folder should appear, for example, **busybox.14.1** or **boa.1.4.1**.

7. Expand the *package*-*version* folder.

   The **patches** node appears.

8. Right-click the **patches** node and find the Quilt menu.

You can expand the **patches** node to see the patches applied for this package. You can further expand the files that are part of a patch, and you can, for example, select a source file such as **boa.c** and use the Quilt menu's **Annotate File** to annotate it. (See *5.6.3 Using Quilt to View Patch Annotations*, p.135.)

5.6.2 Using the Quilt Commands

Quilt tracks changes made by each patch, which can be applied, un-applied, refreshed, and so on. The patches are managed as a stack and applied incrementally on top of the base directory tree plus all preceding patches. They can be pushed and popped to and from the top of the stack. By default, most commands apply to the topmost patch on the stack.

The Quilt menu entries are GUI enactments of the basic commands that you can find in *installDir*/**wrlinux-2.0/wrlinux/host-tools/share/man/man1/quilt.1**. To make this man page available to the **man** command, edit **/etc/man.config**, or put the path in the **MANPATH** variable. At the command-line level, Quilt commands are similar to CVS commands.

The menu entries include the following, listed alphabetically:

- **Add File to Top Patch –** Add one or more files to the topmost or named patch.
- **Add New File to Top Patch** – Add a new file.
- **Annotate File** – Display an annotated listing of the specified file, showing which patches modify which lines. (You can also hover the cursor over a patch indication in the left margin of the file in the Editor.)
- **Compare With** – Diff the specified file(s) in the topmost or specified patch.
- **Delete Patch** – Remove the specific or topmost patch.

- **Export Patch** – Export an existing or new patch file from your platform project to an external file so that it can be shared by other developers. (You can also import the exported patch file into a different project.)
- **Files** – Display the list of files that the specific or topmost patch changes.
- **Fold Patch** – Integrate a patch into the topmost patch. (Specify the file directly, or browse to it.) The tree shows the patch contents to verify the patch.
- **Import Patch** – Import external patches, following the current top patch.
- **New Patch** – Create a new patch with a specified name and insert it after the topmost patch. (The patchname in the tree specifies whether the patch has been loaded or not.)
- **Patches** – Display the list of patches that modified a specified file.
- **Pop** – Remove one or more patches from the stack of applied patches. Click **Details** to see a list of the patches that have been removed and/or restored as a result of the pop.
- **Push** – Apply one or more patches from the series file. Click **Details** to see a list of which files are patched by the push.
- **Refresh Top Patch** – Refresh a specific patch, or by default the top patch.
- **Reload** – Synchronize information that Workbench has cached, such as from command-line usage of Quilt.
- **Rename Patch** – Rename a specific or topmost patch.
- **Remove File from Patch** – Remove one or more files from the named or topmost patch.

5.6.3 Using Quilt to View Patch Annotations

Workbench supports a patch annotation feature, using one of the facilities provided by Quilt.

1. In the project container, browse to a file below a given patch, then select it.
2. Right-click the file and select **Show Patch Annotations.**

   After a moment, this file should appear.
3. Scroll through the file and observe the colored bars in the left hand gutter.
4. Hover the mouse over one of the colored bars.

After a few seconds, a pop-up box appears, giving the name of the patch that contributed this line.

5. Observe that when one of these pop-up boxes appear, some number of boxes also appear in the gutter on the right. These boxes indicate the other locations in this file that were contributed by this patch.

5.6.4 Using Quilt to Import Source and SRPM Packages

SRPM means "Source Redhat Package Manager (archives)." SRPMs are packages than contain the source and spec build rules, rather than just the binary content that regular RPMs contain.

You can also use this procedure to import other packages as well.

To import source packages:

1. Create a platform project.
2. Open its User Space Configuration node, and use the **Import Package** button and script to do the first part of the new package preparation:
   a. Select a classic or SRPM package, for example: **http://download.fedora.redhat.com/pub/fedora/linux/releases/7/Fedora/source/SRPMS/dos2unix-3.1-28.fc7.src.rpm**
   b. Click **Go**.
   c. Click **Close**.

NOTE: You have to close this tool before other Workbench actions, like builds or Quilt actions can occur.

3. Since this is an SRPM, you need to update the spec file, and create a patch file for it:
   a. Select the new package, and click **Patch** in the Targets tab.
   b. Add the **Build directory** link, and refresh if needed.
   c. Right-click anywhere in the **dos2unix** tree, select **Quilt > New Patch**, and call it **dos2unix-wr-integration.patch**.

      This is just an example.
   d. Browse to the file **build/dos2unix-3.1-28.fc7/dos2unix.spec**.

e. Right-click this file, select **Add File to Top Patch**, then click **OK** to accept the change.

f. Make the following changes to **dos2unix.spec** in the Workbench text editor, and save the file:

- Immediately after every **%build** and **%install** section header, add the RPM macro, **%configure_target**. There is one **%build** and one **%install** in this example.
- Optionally, add a change indicator (such as **-WR**), to the Release line.
- Optionally, add an entry to changelog.

g. Right-click, select **Quilt > Refresh top patch**, and click **OK** to accept the refresh.

Observe in **build/dos2unix-3.1-28.fc7/patches** the new file **dos2unix-wr-integration.patch**, which contains the changes.

4. Return to the User Space Configuration node, select the **dos2unix** entry, and click the **Build** button in the Targets tab.

This will build the package and its RPM files.

5. Copy the new patch file back to **dist/dos2unix/patches**, and update the **patches.list** file.

5.6.5 Exporting and Importing Patches with Workbench

You can use the Export Patch feature to export an existing or new patch file from your platform project to an external file so that it can be shared by other developers. It can also be imported into a different project, using the Import Patch feature.

To export a patch from the build system:

1. Open a platform project.
2. Open the project's **build** folder. If that folder is not currently visible, right-click the project and select **Add Links to Development Folders**.
3. Open the folder of the desired package, then open the **patches** folder.
4. Select a registered patch file. If this patch is not up-to-date, right-click and select **Quilt > Refresh Top Patch.**

5. Right-click the patch file, select **Quilt > Export Patch**, and choose a location to place this exported patch.

To import a patch to the build system:

1. Open a different platform project, one without the exported patch of the previous procedure.
2. Open the project's **build** folder. If that folder is not currently visible, right-click the project and select **Add Links to Development Folders**.
3. Open the folder of the desired package, then open the **patches** folder.
4. Right-click the project, select **Quilt > Import Patch**, and select the exported patch.
5. Observe that this patch file is added to the package's patch list, and put into this **patches** folder.

5.7 Automating Target Deployment

You can reboot your target with your latest kernel and file system builds using the **deploy** target in the Project Explorer.

Edit the *installDir*/**wrlinux-2.0/wrlinux/deploy.conf.in** file to include the values as shown in Table 5-1.

Table 5-1 **deploy.conf.in Values**

Field and Value	Description
USER = validUser	Substitute your user name for **validUser**. You will be prompted for the root password during deployment.
TARGET_FS_DIR = /nfsRoot/targets/safeToDelete/ testTarget/fs	Replace the provided path with the path on the host that the target mounts as its root file system. The path *must* contain the string **/safeToDelete**.

Table 5-1 **deploy.conf.in Values**

Field and Value	Description
TARGET_SHUTDOWN_COMMAND = echo ssh targetServer /targetCmds/targetShutdown	Replace **targetServer** with the name or IP address of your target server, and replace **/targetCmds/targetShutdown** with the full path to the command you use to shut down the target.
TARGET_REBOOT_COMMAND = echo ssh targetServer /targetCmds/targetReboot	Replace **targetServer** with the name or IP address of your target server, and replace **/targetCmds/targetReboot** with the full path to the command you use to reboot your target.

The values you enter in the **deploy.conf.in** file are used by the **deploy.sh.in** script. For example in the following line in **deploy.sh.in**:

```
$TARGET_SHUTDOWN_COMMAND $TARGET_ID || exit 1
```

Your shutdown command and target name will be substituted. Refer to the comments and commands in the **deploy.sh.in** script if you would like any further information on how these values are used.

Once you have configured the **deploy.conf.in** file you can easily deploy each new file system build to your target as follows:

1. Select **deploy** in the Project Explorer.
2. Right-click in the Project Explorer and select **Build Target**.

Your target will be deployed with the new file system.

PART III

Projects

6 Projects Overview 143

7 Creating User-Defined Projects 151

8 Creating Native Application Projects 155

6

Projects Overview

6.1 Introduction 143

6.2 Workspace and Project Location 144

6.3 Creating New Projects 145

6.4 Overview of Preconfigured Project Types 146

6.5 Projects and Project Structures 148

6.1 Introduction

Workbench uses *projects* as logical containers. They are also building blocks that can be linked together to create a software system. For example, the Project Explorer lets you visually organize projects into structures that reflect their inner dependencies, and therefore the order in which they are compiled and linked.

Pre-configured templates for various project types let you create or import projects using simple wizards that need only minimal input.

6.2 Workspace and Project Location

By default, your workspace directory is created within your Workbench installation directory. New projects are created in a subdirectory of the workspace directory, named with the project name. For Wind River Platform projects, an additional subdirectory named with the project name and a **_prj** extension is created (see *3.3.3 Understanding What Else You Can Build*, p.58).

Wind River Workbench cannot know where your source files are located, so it initially suggests a default workspace directory within the installation directory. However, this is not a requirement, or even necessarily desirable. If you use a workspace directory outside of the Workbench installation tree, you can preserve the integrity of your projects after product upgrades or installation modifications.

Normally, you would set your workspace directory at the root of your existing source code tree and create your Workbench projects there. For multiple, unrelated source code trees, you can use multiple workspaces.

Some considerations when deciding where to create your project:

Create project in workspace
Leave this selected if you want to create the project under the current workspace directory. This is typical for:

- Projects created from scratch with no existing sources.
- Projects where existing sources will be imported into them later on (for details, see *9.3 Adding Application Code to Projects*, p.162).
- Projects where you do not have write permission to the location of your source files.

Create project at external location
Select this option, click **Browse**, then navigate to a different location if you want to create the project outside the workspace. This is typical for:

- Projects being set up for already existing sources, removing the need to import or link to them later on.
- Projects being version-controlled, where sources are located outside the workspace.

Create project in workspace with content at external location
Select this option, click **Browse**, then navigate to your source location if you do not want to mix project files with your sources, or copy sources into your workspace. This is useful for:

- Projects where you do not have write permission to the location of your source files.
- Projects where team members have their own projects, but share common (sometimes read-only) source files. This option eliminates the need to create symbolic links to your external files before you can work with them in Workbench.

6.3 Creating New Projects

Although you can create projects anywhere, you would generally create them in your workspace directory (see *6.2 Workspace and Project Location*, p.144). If you follow this recommendation, there will generally be no need to navigate out of the workspace when you create projects. Note that if you do create projects outside the workspace, you must have write permission at the external location because Workbench project administration files are written to this location.

To create a new project, you can use any of the following options:

- Click the **New** toolbar icon
- Select **File > New > Wind River Workbench Project** to open the New Wind River Workbench Project wizard to help you create one of the pre-configured project types.
- Click the toolbar icon
- Select **File > New >** *ProjectType*.

For more information about projects types, see *Overview of Preconfigured Project Types*, p.146.

To create one of the demonstration sample projects, select **File > New > Example** to open the New Example wizard. Each comes with instructions explaining the behavior of the program.

Whichever menu command you choose, a wizard will guide you through the process of creating the specific type of project you select.

6.3.1 Subsequent Modification of Project Creation Wizard Settings

All project creation wizard settings can be modified in the Project Properties once the project exists. To access the Project Properties from the Project Explorer, right-click the icon of the project you want to modify and select **Properties**. For more information about project properties, see *11.4 Accessing Build Properties*, p.196.

6.3.2 Projects and Application Code

All application code is managed by projects of one type or another. You can import an existing Workbench-compatible project as a whole, or you can add new or existing source code files to your projects. For more information, select **File > Import** to open the Import File dialog, then press the help key for your host.

6.4 Overview of Preconfigured Project Types

Workbench offers the following pre configured project types:

- *Wind River Linux Application Project*, p.147
- *Wind River User-Defined Project*, p.147
- *Wind River Linux Platform Project*, p.147
- *Wind River Linux Kernel Module Project*, p.147
- *Customer Specific Linux Application Project*, p.147
- *Customer Specific Linux Kernel Project*, p.148
- *User-Defined Project*, p.148
- *Native Application Project*, p.148

NOTE: You may see different project types depending on your installed software. Platform developers have access to kernel source and kernel tools, whereas application developers do not. Refer to the documentation on the particular software for details on specific project types.

Wind River Linux Application Project

Wind River Linux application projects are developed on the host computer and deployed on the target. These projects dynamically generate build specs for Wind River Linux-supported board architectures and kernel and file system combinations. See *2.3 Creating a Cross-Debug Mode Project*, p.37 for more information on Wind River Linux application projects.

Wind River User-Defined Project

Wind River user-defined projects run on the Wind River Linux Platform, with a user-defined build based on an existing Makefile. You can specify the project tree structure and project references according to specific projects that you have already defined, which are then referenced as subprojects. The makefiles for user-defined projects can still use values from the Wind River Linux build specs, to help set the correct cross compile tool chain and architecture from the respective target's sysroot.

Wind River Linux Platform Project

Wind River Linux platform projects are developed on the Linux host computer and deployed on pre-defined targets. Platform projects can include prebuilt or customized kernels, and pre-defined or customized file systems.Workbench provides special support for Wind River platform projects including kernel and user space configuration tools, and multiple build targets. See *3. Creating and Debugging Platform Projects* for more details.

Wind River Linux Kernel Module Project

Wind River Linux kernel module projects allow you to specify superprojects as well as subprojects. You can specify a build (make) command; suffixes for source files, kernel root directory, target architecture; and a cross-compile prefix.

Customer Specific Linux Application Project

Customer Specific Linux application projects are built and reside on the host computer, but run on the target kernel. For these projects, Workbench basically provides a generic managed build allowing you to derive your specific build specs

from a set of predefined ones. See 2.3 Creating a Cross-Debug Mode Project, p.37, for a tutorial on building this type of project.

NOTE: To enable Customer Specific Linux projects: select **Window > Preferences > Wind River > Capabilities**. Expand **Development**. Check **Customer Specific Linux Development**. Click **Apply**.

Customer Specific Linux Kernel Project

Customer Specific Linux *kernel* projects let you configure (customize or scale) and build a Linux kernel to run on the target. These projects basically wrap the typical Linux kernel CLI build. Because your subsequent Linux application projects will run on the Linux kernel on the target, this is often a necessary first project to build, unless you already have a target kernel.

User-Defined Project

User-defined projects do not use Workbench build support or pre-configured features. These projects can be anything and it is up to the user to organize and maintain the build system, target file system population, and so forth.

Native Application Project

A native application project is built and run on the host computer. In effect, your Workbench host serves as the target. See 2.2 Running Hello World in a Native-Mode Linux Environment, p.32 for an example of building a native application project.

6.5 Projects and Project Structures

All individual projects of whatever type are self-contained units that have no inherent relationship with any other projects. The system is initially flat and unstructured. You can, however, construct hierarchies of project references within Workbench. These hierarchies will reflect inter-project dependencies and therefore also the build order.

When you attempt to create such hierarchies of references, this is validated by Workbench; that is, if a certain project type does not make sense as a subproject of some other project type, or even the same project type, such a reference will not be permitted.

6.5.1 Adding Subprojects to a Project

Workbench provides the following ways to create a subproject/superproject structure:

- You can use the **Add as Project Reference** dialog. In the Project Explorer, right-click the project that you want to make into a subproject and choose **References > Add as Project Reference**, or select the project and choose **Project > Add as Project Reference**. In the dialog, you will see a list of valid superprojects; you can select more than one.
- You can use the **Project References** page in the **Properties** dialog. In the Project Explorer, right-click the project that you want to make into a *super*project and choose **Properties**, or select the project and choose **Project > Properties**. Then select **Project References**. In the dialog, you will see a list of projects; select the ones that you want to make into subprojects.

Subprojects appear as subnodes of their parents (superprojects).

Workbench validates subproject/superproject relationships based on project type and target operating system. It does not allow you to create certain combinations. In general, a user-defined project can be a subproject or superproject of any other project with a compatible target operating system.

6.5.2 Removing Subprojects

To undo a subproject/superproject relationship, use one of these methods:

- In the Project Explorer, right-click the subproject and choose **References > Remove Project Reference**, or select the subproject and choose **Project > Remove Project Reference**.
- In the Project Explorer, right-click the superproject and choose **Properties**, or select the superproject and choose **Project > Properties**. Then select **Project References** and clear the subprojects that you want to disassociate from their current parent.

7 Creating User-Defined Projects

7.1 Introduction 151

7.2 Creating and Maintaining Makefiles 152

7.3 Creating User-Defined Projects 152

7.4 Configuring User-Defined Projects 153

7.1 Introduction

User-Defined Projects assume that you are responsible for setting up and maintaining your own build system, file system population, and so on. The user interface provides support for the following:

- You can configure the build command used to launch your build utility; this allows you to start builds from the Workbench GUI. You can also configure different rules for building, rebuilding and cleaning the project.
- You can create build targets in the Project Explorer that reflect rules in your makefiles; this allows you to select and build any of your make rules directly from the Project Explorer.
- Build output is captured to the Build Console.

7.2 Creating and Maintaining Makefiles

When you create a User-Defined project, Workbench checks the root location of the project's resources for the existence of a file named **Makefile**.

If you used the **-f** make option to specify a different filename in the New Project wizard's Build Command field, you could include a relative or absolute path to a subdirectory.

If the file does not exist, Workbench creates a skeleton makefile with a default **all** rule and a **clean**. This lets you use the **Build Project**, **Rebuild Project**, and **Clean Project** menu commands. It also prevents the generation of build errors. But, you are responsible for maintaining this makefile, and you can write any other rules into it at any time.

If you base your User-Defined project on an existing project, that project's makefile is copied to the new project, overwriting any already there. (If necessary, you can change the name of the new project's makefile using the **-f** make option to avoid overwriting an existing makefile.)

7.3 Creating User-Defined Projects

Before creating the project, see the general comments on projects and project creation in *6. Projects Overview*.

1. Create a User-Defined project by selecting **File > New > User-Defined Project**.
2. Select a target operating system, then click **Next**.
3. Type a name for your project.
4. Decide where to create your project:

   Create project in workspace
   Leave this selected to create the project in the current workspace directory.

   Create project at external location
   Select this option, click **Browse**, then navigate to a different location to create the project outside the workspace.

Create project in workspace with content at external location
Select this option, click **Browse**, then navigate to your source location if you do not want to mix project files with your sources, or to copy sources into your workspace.

5. When you are ready, click **Finish**. Your project appears in the Project Explorer.

7.4 Configuring User-Defined Projects

After you create your project, configure its build targets, specs, and macros.

For general details about build properties, see *11.4 Accessing Build Properties*, p.196 or press the help key for your host.

1. To access build properties for your project, right-click it in the Project Explorer and select **Properties**.
2. From the **Properties** dialog, click **Build Properties**.

NOTE: Build tools and build paths cannot be configured for user-defined projects.

The makefiles for user-defined projects can use values from the build specs to help set the correct cross-compile toolchain and architecture from the target's sysroot.

7.4.1 Configuring Build Support

Use this tab to configure build support for your project:

1. Build support is enabled by default. Click **Disabled** to disable it, and click **User-defined build** to re-enable it.
2. If necessary, edit the default build command.
3. Specify whether received build targets should be passed to the next level.
4. Specify when Workbench should refresh the project after a build.

   Because a refresh of the entire project can take some time (depending on its size), Workbench does not do it by default. You may choose to refresh the current project, the current folder, the current folder and its subfolders, or nothing at all. This option applies to all build runs of the project.

7.4.2 Configuring Build Targets

Use this tab to configure make rules and define custom build targets for your project.

1. Type the desired make rules into the fields in the **Make rules** section. These rules are run when you select the corresponding options from the **Project** menu or when you right-click your project in the Project Explorer and select them from the context menu.

   The **Build Folder** and **Clean Folder** options are available when you select a folder in the Project Explorer.

2. To define a custom build target, click **New**. The **New Custom Build Target** dialog opens.

3. Type in a name for your build target, then type in the make rule or external command that Workbench should execute. You can also click **Variables** and add a context-sensitive variable to the make rule or command.

   The variables represented in the **Select Variable** dialog are context-sensitive, and depend on the current selection in the Project Explorer. For variables that contain a file-specific component, the corresponding target is only enabled when a file is selected and the variable can be evaluated. Build targets without file-specific components are always enabled.

4. Choose the type, whether it is a **Rule** or a **Command**.

5. Choose a refresh option for the build target, specifying whether Workbench should use the project setting, refresh the current folder or project, or do nothing. Click **OK** to close the dialog.

6. Edit a build target's options by clicking **Edit** or **Rename**. You can also edit the options (except name) by clicking in the column itself.

7. Continue configuring your project or click **OK** to close the Build Properties.

Once you have defined a build target, it is available when you right-click a project and select **Build Options**. The build targets are inherited by each folder within the project, eliminating the need to define the same build targets in each individual folder.

8 Creating Native Application Projects

8.1 Introduction 155

8.2 Creating a Native Application Project 156

8.3 Application Code for a Native Application Project 158

8.1 Introduction

Use a *Native Application project* for C/C++ applications developed for your host environment.

Workbench provides build and source analysis support for native GNU and Microsoft development utilities (assembler, compiler, linker, and archiver) though you must acquire and install these utilities as they are not distributed with Workbench.

NOTE: Workbench supports the MinGW, Cygnus, and MS DevStudio compilers. Compilers for native development are distributed with Wind River Platforms, but not with Workbench.

You may use the GDB debugger built into Eclipse to debug native-mode, self-hosted applications such as Hello World. For more information, see *2.2.3 Debugging Self-Hosted Applications with the CDT/GDB Debugger*, p.35.

8.2 Creating a Native Application Project

Before creating the project, please take a look at the general comments on projects and project creation in *6. Projects Overview*.

To create a Native Application project:

1. Choose **File > New > Native Application Project**.

   The **New Native Application Project** wizard appears. If you have multiple operating systems installed, you are asked to select a target operating system. If you see this field, select a version from the drop-down list and click **Next**.

2. Enter a **Project name** and **Location**.

   If you choose **Create project in workspace** (default) the project will be created under the current workspace directory. If you choose to **Create project at external location**, you can navigate to a location outside the workspace (see also *6.2 Workspace and Project Location*, p.144 and *6.3 Creating New Projects*, p.145).

   The project appears in the Project Explorer. To see the project location, right-click the project and select **Properties**, then select the **Info** node of the Properties dialog.

   When you are ready, click **Next**.

3. If you have created other projects, you are asked to define the project structure (the super- and subproject context) for the project you are creating.

   The text beside the **Link to superproject** check box refers to whatever project is currently highlighted in the Project Explorer (if you do not see this check box, no valid project is highlighted). If you select the check box, this will be the superproject of the project you are currently creating.

   The check boxes in the **Referenced subprojects** list represent the remaining projects in the workspace that can be validly referenced as subprojects by the project you are currently creating.

   When you are ready, click **Next**.

NOTE: All settings in the following wizard pages are build related. You can therefore verify/modify them after project creation in the Build Properties node of the project's Properties, see *11. Building Projects*.

4. A Native Application project is a predefined project type that uses Workbench **Build support**, so you can only select either this, or no build support at all. If

you are creating a project because you want to browse symbol information only and you are not interested in building it, you could also disable build support.

The **Build command** specifies the make tool command line.

Build output passing: If the project is a subproject in a tree, its own objects (implicit targets) as well as the explicit targets of its subprojects, can be passed on to be linked into the build targets of projects that are further up in the hierarchy.

When you are ready, click **Next**.

5. **Build Specs**: The list of available build specs will always be available. By checkmarking individual specs, you enable them for the current project, which means that you will, in normal day to day work, only see relevant (enabled) specs in the user interface, rather than the whole list.

   If you are working on a Windows application, you would normally enable the **msvc_native** build spec, and disable the **gnu-native** build specs. If you are working on a Linux or Solaris native application, you would normally enable the GNU tool version you are using, and disable all others.

   The **Debug Mode** check box specifies wether or not the build output includes debug information.

   When you are ready, click **Next**.

6. **Build Target**: The **Build target name** is the same as the project name by default. You can change the name if necessary, but if you delete the contents of the field, no target will be created.

   Build tool: For a Native Application project you can select:

   - **Linker**: This is the default selection. The linker produces a a *BuildTargetName* (**.exe** for Windows native projects) executable file.

     The **Linker** output product cannot be passed up to superprojects, although the current project's own, unlinked object files can, as can any output products received from projects further down in the hierarchy (see step 4. above).

   - **Librarian**: This is the default selection if you specified that the project is to be linked into a project structure as a subproject. The Librarian produces a *TargetName***.a** (or **.lib** for Windows native projects) archive file**.**

The **Librarian** output product can be passed up to superprojects, as can the current project's own, unlinked object files, as well as any output products received from projects further down in the hierarchy (see step 4. above).

7. When you are ready, you can review your settings using the **Back** button or click **Finish**.

   The Native Application project is created and appears in the Project Explorer, either at the root level, or linked into a project tree, depending on your selection in step 3. above.

8.3 Application Code for a Native Application Project

After project creation you have the infrastructure for a Native Application project, but often no actual application code. If you are writing code from the beginning, you can add new files to a project. If you already have source code files, you will want to import these to the project. For more information, refer to *Importing Resources*, p.162, and *Adding New Files to Projects*, p.163.

PART IV

Development

9 Working in the Project Explorer 161

10 Navigating and Editing 169

11 Building Projects 189

12 Building: Use Cases 201

9 Working in the Project Explorer

9.1 Introduction 161

9.2 Creating Projects 162

9.3 Adding Application Code to Projects 162

9.4 Opening and Closing Projects 163

9.5 Scoping and Navigation 164

9.6 Moving, Copying, and Deleting Resources and Nodes 165

9.1 Introduction

The Project Explorer is your main graphical interface for working with projects. You use the Project Explorer to create, open, close, modify, and build projects. You also use it to add or import application code, to import, or customize build specifications, and to access your version control system.

Various filters, sorting mechanisms, and viewing options help to make project management and navigation more efficient. Use the arrow at the top right of the Project Explorer to open a drop-down menu of these options.

9.2 Creating Projects

Creating projects is discussed in general under *6.3 Creating New Projects*, p.145. Specific descriptions for creating individual project types are provided in the other chapters in *Part III. Projects*.

9.3 Adding Application Code to Projects

After creating a project, you have the infrastructure for a given project type, but no actual application code. If you already have source code files, you will want to import these to the project.

Importing Resources

You can import various types of existing resources to (newly created) projects by choosing **File > Import**.

For details about the entries in the Import File dialog, see *Wind River Workbench User Interface Reference: Import File Dialog*.

NOTE: Importing resources creates a link to the location of those resources; it does not copy them into your workspace.

Later, if you want to delete a project, check the path in the **Confirm Project Delete** dialog very carefully when deciding whether to choose **Also delete contents under** '*path*' or **Do not delete contents**—choosing to delete the project contents may delete your original sources or the contents of a project in a different workspace, rather than the project in your current workspace.

Adding New Files to Projects

To add a new file to a project, choose **File > New > File**.

You are asked to **Enter or select the parent folder**, and to supply **Filename**.

For a description of the **Advanced** button, and what it reveals, press **F1** and select **New file wizard**.

9.4 Opening and Closing Projects

You can open or close a project by selecting it in the tree and choosing **Project > Open** (if it is currently closed), or **Project > Close** (if it is currently open). You can also use the corresponding commands on the Project Explorer's right-click context menu.

When you close a project:

- The icon changes to its closed state (by default grayed) and the tree collapses.
- All project member files that are open in the editor are closed.
- All subprojects that are linked exclusively to the closed project are closed as well. However, subprojects that are shared among multiple projects remain open as long as a parent project is still open, but can be closed explicitly at any time.
- In general, closed projects are excluded from all actions such as symbol information queries, and from workspace or project structure builds (that is, if a parent project of a closed subproject gets built).
- It is not possible to manipulate closed projects. You cannot add, delete, move, or rename resources, nor can you modify properties. The only possible modification is to delete the project itself.
- Closed projects require less memory.

9.5 Scoping and Navigation

There are a number of strategies and Workbench features that can help you manage the projects in your workspace, whether you are working with multiple projects related to a single software system, or multiple unrelated software systems.

- **Close projects**

  If you expect to be working in a different context (under a different root project) for a while, you can select the root project you are leaving, and right-click **Close Project**.

  If you close your root projects when you stop working on them, you will see just the symbols and resources for the project on which you are currently working (see also *9.4 Opening and Closing Projects*, p.163).

- **Go into a project**

  If you want to see, for example, the contents of only one software system in the Project Explorer, select its root project node and right-click **Go Into**. You can then use the navigation arrows at the top of the Project Explorer to go back out of the project you are in, or to navigate history views.

- **Open a project in a new window**

  If you expect to be switching back and forth between software systems (or other contexts) at short intervals, and you do not want to change your current configuration of open editors and layout of other views, you can open the other software system's root project in a new window (right-click **Open in New Window**). This essentially does the same as **Go Into** (see **Go Into a Project**), except that a new window is opened, thereby leaving your current Workbench layout intact.

- **Open a new window**

  You can open a new window by choosing **Window > New Window**. This opens a new window to the same workspace, leaving your current Workbench window layout intact while you work on some other context in the new window.

- **Use Working Sets**

  You can use working to set the scope for all sorts of queries. You can, for example, create working sets for each of your different software systems, or any constellation of projects, and then scope the displayed Project Explorer content (and other query requests) using the pull-down at the top-right of the Project Explorer.

  To create a Working Set, from the drop-down menu, choose **Select Working Set**. In the dialog that appears, click **New**, then, in the next dialog, specify the **Resource** type.

  In the next dialog select, for example, a software-system root project and give the working set a name. When you click **Finish**, your new working set will appear in the **Select Working Set** dialog's list of available working sets.

  After the first time you select a working set in the **Select Working Set** dialog, the working set is inserted into the Project Explorer's drop-down menu, so that you can directly access it from there.

- **Use the Navigate Menu**

  For day-to-day work, there is generally no absolute need to see the contents of your software systems as presented in the Project Explorer.

  Using the **Navigate > Open Resource** (to navigate files) and **Navigate > Open Symbol** (to jump straight to a symbol definition) may often prove to be the most convenient and efficient way to navigate within, or among, systems.

9.6 Moving, Copying, and Deleting Resources and Nodes

The resources you see in the Project Explorer are normally displayed in their logical, as opposed to physical, configuration (see *6.5 Projects and Project Structures*, p.148). Depending on the type of resource (file, project folder) or purely logical element (target node) you are manipulating, different things will happen. The following section briefly summarizes what is meant by resource types and logical nodes.

9.6.1 Understanding Resources and Logical Nodes

Resources is a collective term for the *projects*, *folders*, and *files* that exist in Workbench.

There are three basic types of resources:

Files
Equivalent to files as you see them in the file system.

Folders
Equivalent to directories on a file system. In Workbench, folders are contained in projects or other folders. Folders can contain files and other folders.

Projects
Contain folders and files. Projects are used for builds, version management, sharing, and resource organization. Like folders, projects map to directories in the file system. When you create a project, you specify a location for it in the file system.

When a project is open, the structure of the project can be changed and you will see the contents. A discussion of closed projects is provided under *9.4 Opening and Closing Projects*, p.163.

Logical nodes is a collective term for nodes in the Project Explorer that provide structural information or access points for project-specific tools.

Subprojects
A project is a resource in the root position. A project that references a superproject is, however, a logical entity; it is a reference only, not necessarily (or even normally) a physical subdirectory of the superproject's directory in the file system.

Build Target Nodes
These are purely logical nodes to associate the project's build output with the project.

Tool Access Nodes
These allow access to project-specific configuration tools.

9.6.2 Manipulating Files

Individual files, for example source code files, can be copied, moved, or deleted. These are physical manipulations. For example, if you hold down **CTRL** while you

drag-and-drop a source file from one project to another, you will create a physical copy, and editing one copy will have no effect on the other.

9.6.3 Manipulating Project Nodes

Although copying, moving, or deleting project nodes are undertaken with the same commands you would use for normal files, the results are somewhat different because a project folder is a semi-logical entity. That is, a project is a normal resource in the root position. A project that is referenced as a subnode is, however, a logical entity; it is a reference only, not a physical instance.

Moving and (Un-)Referencing Project Nodes

If you select a project folder node and right-click **Move**, you will be asked to enter (browse for) a new file system location. All the files associated with the current project will then be physically moved to the location you select, without any visible change in the Project Explorer (you can verify the new location in the **Project Properties**).

When you use right-click a project node and choose **Add as Reference** (or, if you have selected a subproject, right-click and choose **Remove Reference**), a dialog opens to allow you to place the currently selected project reference into other projects as a subproject, or, if you are using **Remove Reference**, to remove the currently selected project from its structural (logical) context as a subproject (in which case it will be moved to the root level as a standalone project in the Project Explorer).

Deleting Project Nodes

To delete a subproject, which might potentially be linked into any number of other project structures, you first have to either unlink (right-click it and press **Delete**) all instances of the subproject, or get a *flat* view of your workspace. To do this, open the drop-down list at the top-right of the Project Explorer's toolbar and choose **Hide > Project Structure**. This hides the logical project organization and provides a flat view with a single instance of the (sub)project that you can then delete by pressing **Delete** again.

When you delete a project you are asked whether or not you want to delete the contents. If you choose not to delete the contents, the only thing that happens is

that the project (and all its files) are no longer visible in the workspace; there are no file system changes.

9.6.4 Manipulating Target Nodes

Target nodes cannot be copied or moved. These are purely logical nodes that make no sense anywhere except in the projects for which they were created. If you copy or move entire projects, however, the target nodes and generated build-targets beneath them are also copied.

Deleting a target node also removes the convenience node that represents the generated, physically existing build-target. However, the physically existing build-target (if built) is not deleted from the disk.

The convenience node referred to above, lets you see at a glance whether the target has been built or not, even if you have uncluttered your view in the Project Explorer by hiding build resources (in the drop-down menu at the top-right choose **Hide > Build Resources**) and/or collapsing the actual target node. If you have collapsed the node, the **+** sign will indicate that the build-target exists).

10 Navigating and Editing

10.1 Introduction 169

10.2 Wind River Workbench Context Navigation 170

10.3 The Editor 173

10.4 Search and Replace 175

10.5 Configuring the Indexer for Source Analysis 176

10.1 Introduction

Workbench navigation views allow seamless cross-file navigation based on symbol information. For example, if you know the name of a function, you can navigate to that function without worrying about which file it is in. You can do this either from an editing context, or starting from the *The Symbol Browser*, p.171. On the other hand, if you prefer navigating within and between files, you can use the *The File Navigator*, p.172.

Source analysis is the parsing and analysis of source code symbol information. This information is used to provide code editing assistance features such as multi-language syntax highlighting, code completion, parameter hints, and definition/declaration navigation for files within your projects.

Apart from the things you see directly in the Editor, source analysis also provides the data for code comprehension and navigation features such as include browsing, call trees, as well as resolving includes to provide the compiler with include search paths.

NOTE: Syntax highlighting is provided for file system files that you open in the Editor, but no other source analysis features are available for files that are outside your projects.

10.2 Wind River Workbench Context Navigation

Various filters are available on each tool's local toolbar. Hover the mouse over each button to see a tool tip describing what it does. At the top-right, a pull-down menu provides additional filters, including *working sets* (if you have defined any). An active working set is marked by a bullet next to its name in the pull-down menu.

Generally, you will want to navigate to symbols, or analyze symbol-related information, from an Editor context. The entry points are as follows:

- The right-click context menu of a symbol
- Keyboard shortcuts that act on the current selection in the Editor:

  F3 – Jump between associated code, for example, between definition/declaration or function definition/call. There is no navigation from workspace files to external files, i.e. files outside your projects.

  F4 – Open the type hierarchy of the current selection (see *Type Hierarchy View*, p.172).

  CTRL+ALT+H – Open the call tree of the current selection (see *Wind River Workbench User Interface Reference: Call Tree View*).

  CTRL+I – Open the include browser to view the includes of the current selection (see *Include Browser*, p.173).

- Keyboard shortcuts that open dialogs from which you can access symbols in any of your projects:

  SHIFT+F3 – Display the **Open Symbol** dialog.

  SHIFT+F4 – Display the **Open Type Hierarchy** dialog.

ALT+SHIFT+H – Display the **Open Call Tree** dialog.

CTRL+SHIFT+R – Displays the **Open Resource** dialog.

These options are also available from the **Navigate** toolbar menu.

The Symbol Browser

By default, the Symbol Browser is a tab in the left pane of the main window, together with the Project Explorer.

Use the Symbol Browser for global navigation. Because the Symbol Browser presents a flat list of all the symbols in all the open projects in your workspace, you might want to constrain the list by using *Working Sets*. You can configure and select working sets using the Project Explorer's local pull-down menu.

In addition, very large symbol loads can cause delays of up to several minutes while Workbench loads the symbols. Loading smaller batches of symbols can decrease this delay. Specify the size of the symbol batch using the Preferences dialog. For more information, see *Wind River Workbench User Interface Reference: Debug View*.

Text Filtering

The **Name Filter** field at the top of the view provides match-as-you-type filtering. The field also supports wild cards: type a question mark (**?**) to match any single letter; type an asterisk (*****) to match any number of arbitrary letters. Selecting **Hide Matching** next to the **Name Filter** field inverts the filter you entered in the field, so you see only those entries that do not match your search criteria.

For a guide to the icons in the Symbol Browser, see *Wind River Workbench User Interface Reference: Symbol Browser View*.

The Outline View

The Outline view is to the right of the currently active Editor, and shows symbols in the currently active file.

Use the Outline view to sort, filter, and navigate the symbols in the context of the file in the currently active Editor, as well as to navigate out of the current file context by following call and reference relationships.

For a guide to the icons in the Outline view, see *Wind River Workbench User Interface Reference: Outline View*.

The File Navigator

If you have never used the File Navigator, you can open it by choosing **Window > Show View > Other**. In the dialog that opens, select **Wind River Workbench > File Navigator** and click **OK**. After the first time you open the File Navigator, a shortcut appears directly under the **Window > Show View** menu. By default, the File Navigator appears as a tab at the left of the Wind River Workbench window, along with the Project Explorer and the Symbol Browser.

The File Navigator presents a flat list of all the files in the open projects in your workspace, so you can constrain the list by using *Working Sets*. You can configure and select working sets using the File Navigator's local pull-down menu.

The left column of the File Navigator shows the file name, and is active; double-clicking a file name opens the file in the Editor, and right-clicking a file allows you to compile the file and build the project, among other tasks. The right column displays the project path location of the file.

The **File Filter** field at the top of the view works in the same way as the **Name Filter** field in the Symbol Browser, see *The Symbol Browser*, p.171.

Type Hierarchy View

Use the Type Hierarchy view to see hierarchical **typedef** and type-member information.

To open the Type Hierarchy view:

- Right-click a symbol in the Editor, Outline, or Symbol Browser view and select **Type Hierarchy view**.
- Click the toolbar button on the main toolbar.
- Select **Navigate > Open Type Hierarchy**.

For more information, see the *Wind River Workbench User Interface Reference: Type Hierarchy View*.

Include Browser

By default, the Include Browser appears as a tab at the bottom-right.

To open the Include Browser:

- Right-click a symbol in the Editor, Outline, or Symbol Browser view and select **Open Include Browser**.
- Right-click a file in the File Navigator or the Project Explorer and select **Include Browser**.
- Select **Navigate > Open Include Browser**.

Use the Include Browser to see which file includes, or is included by, the file you are examining. Use the buttons on the Include Browser's local toolbar to toggle between showing include and included-by relationships. Double-click an included file in the Include Browser to open the file in the Editor at the include statement.

10.3 The Editor

The Editor is your primary view for editing and debugging source code. The Editor is language-aware, and can parse C, C++, Ada, and Assembler files. Many Editor features are configurable in the Preferences (see *Wind River Workbench User Interface Reference: Editor*).

Code Templates

The Editor uses templates to extend code assist (shortcut **CTRL+SPACE**) by inserting recurring patterns of text.

In the case of source code, common patterns are **for** loops, **if** statements and comment blocks. Those patterns can be parameterized with variable placeholders that are resolved and substituted when the template is inserted into the text. Unresolved variables can be link-edited after inserting the template, which means that the first unresolved variable is selected, and all occurrences of this variable are edited simultaneously when you enter the correct text.

An example template might look like the following:

```
for (int ${var} = 0; ${var} < ${max}; ++${var}) {
    ${cursor}
}
```

Provided Templates

Workbench provides the following templates. Auto-insert is turned on by default.

Name	Description
author	author name
catch	catch block
class	class declaration
comment	default multiline comment
do	do while statement
else	else block
elseif	else if block
for	for loop
for	for loop with temporary variable
if	if statement
ifelse	if else statement
main	main method
namespace	namespace declaration
new	create new object
stderr	print to standard error
stdout	print to standard output
switch	switch case statement
try	try catch block
using	using a namespace

Many template options are configurable in the Preferences (see *Wind River Workbench User Interface Reference: Editor*).

Configuring a Custom Editor

Workbench has a single global mapping between file types and associated editors. This mapping dictates which editor will be opened when you double-click a file in the Project Explorer, or when the debugger stops in a given file.

Configuring the custom editor through file associations will cause the correct editor to be opened, and the instruction pointer to be painted in the editor gutter. To view and modify the mappings, go to **Window Preferences > General > Editors > File Associations**.

NOTE: Some debugger features require additional configuration; for details, see *18.2.1 Configuring Debug Settings for a Custom Editor*, p.289.

10.4 Search and Replace

The Workbench search tool is a fast, index-based global text search/replace tool. The scope of a search can be anything from a single file to all open projects in the workspace. You can query for normal text strings, or regular expressions. Matches can be filtered according to location context (for example, show only matches occurring in comments). Text can be globally or individually replaced, and restored if necessary. You can create working sets from matched files, and you can save and reload existing queries.

Text retrieval is context sensitive to text selected in the Editor. If no text is selected in the Editor, an empty search dialog opens. If text is selected in the Editor, the retrieval is immediately initiated according to the criteria currently defined in the dialog.

To open the search dialog, or to initiate a context sensitive search, use:

- **CTRL+2**
- A scoping option from the global **Search** menu

For more information, open the search dialog and press the help key for your host.

10.5 Configuring the Indexer for Source Analysis

Editing, navigating, and code completion rely on source analysis done by the Indexer.

The resulting index is the basis for any source navigation and editing capabilities, such as navigation between declaration and definition of methods, showing included files in the Include Browser, showing the call tree of functions and methods, showing the type hierarchy of classes, code completion in the editor, hover information for function and method occurrences (a tool tip showing the signature and comments of the related declaration or definition), and so on.

Workbench uses the Eclipse Fast C/C++ Indexer per default. You can, however, switch off source analysis for any project by selecting the **No Indexer** option on the indexer property page. For more information about the Indexer, see the *C/C++ Development User Guide* under **Help > Help Contents**.

10.5.1 Setting Indexer Preferences

You can set basic indexer preferences such as whether to index a project, or to index all files in a project, on either a global (workspace) or per-project basis.

Global (Workspace) Preferences

To set global (workspace) indexer preferences:

1. Select **Window > Preferences > C/C++ > Indexer** to open the preferences dialog.
2. Select **Index all files** if you want to parse all source and header files.

   By default, the only files indexed are those added to a managed project's build target, or, for user-defined projects, those included by enabling build-output analysis.
3. Select **No Indexer** in the drop down, if you want no files to be parsed.
4. Set **Indexing strategy** to **after every file change**, if you do not want the index upated automatically.
5. Accept the **Cache limits** as specified: share of heap memory, absolute file size, and header file cache size.
6. Click **Apply**.

Project-Specific Properties

To set project-specific indexer preferences (properties):

1. Right-click a project in the Project Explorer, then select **Properties > C/C++ General > Indexer**.
2. Select **Enable project specific settings**,.
3. Select **Index all files** if you want to parse all source and header files.

   By default, the only files indexed are those added to a managed project's build target, or, for user-defined projects, those included by enabling build-output analysis.
4. Select **No Indexer** in the drop down, if you want no files to be parsed.
5. Click **Apply**.

10.5.2 Editing Build Properties

Indexing works differently depending on the type of project:

- Managed projects allow full configuration of build settings via the Workbench build properties page. Include paths and symbols (preprocessor macros) defined in the build properties are directly passed to the indexer. The build scope (the files to be built) defines which files the indexer processes by default. Other files are not be parsed by default.
- User-defined projects do not allow full configuration of build settings via the Workbench GUI. Manual editing of makefiles may be required in order to get a fully functional build. The indexer can, however, derive include paths, symbols, and build scope through build output analysis.

For both types of projects, you can define additional include paths and symbols in the Paths and Symbols property page.

Setting Build Properties for Managed Projects

Project creation wizards create default build targets, unless you clear the **Build Target name** field on the Build Target wizard page.

A build target defines the build scope that is used by the indexer to process source code. Files outside this build target are not parsed per default.

If there are no build targets, all files are parsed, otherwise you would end up with an empty index, and no source analysis or navigation.

A build target must contain at least one source file. If you delete all files from a build target, and there are no non-empty build targets in the project, then nothing is parsed, and the source navigation tools such as Type Hierarchy, Call Tree, etc., will not work (correctly).

To create a new build target, select **New > Build Target** from the main menu, or right-click the **Build Targets** node under the project in the Project Explorer and select **New Build Target**. Then follow the instructions in the wizard.

You can use the build properties to control the whole build process. For that purpose, select the project in the Project Explorer view and navigate to the **Build Properties** page. The available tabs are Build Support, Build Targets, Build Specs, and Build Macros. You can use these as follows:

- Create and/or enable proper build specs on the Build Support and Specs tab page.
- Edit build tool settings on the Build Tools tab page, if necessary.
- Edit build macros on the Build Macros tab page. In many cases, the DEFINES build macro is already predefined. Use this to define preprocessor macros (symbols), if available. All symbols (including -D prefix) defined by build macros and passed to build tools are also passed on to the indexer, and thus used for parsing source code of the given resource. For instance, you may set DEFINES to -DMACRO1 -DMACRO2 and pass DEFINES to the C or C++ compiler on the Build Tools tab page, in the Command or Tool Flags section.

  In addition to the symbols defined on the Build Macros tab page, the symbols defined implicitly by the selected build tools are also passed on to the indexer.
- Edit build paths on the Build Paths tab page. Use **Generate** to have include paths determined for unresolved include directives. Build paths of the default build spec are passed on to the indexer, as well as to the build tools.

  In addition to the include paths defined on the Build Paths tab page, the include paths defined implicitly by the selected build tools are also passed on to the indexer.

NOTE: The indexer uses only the symbols and include paths defined for the default build spec to parse source code for a given project.

To generate include paths and resolve include directives for managed projects:

1. Begin to open the Resolve Include directives dialog by choosing one of these paths:
   - Right-click the project in the Project Explorer, select **Build Options > Generate Include Search Paths**.
   - Or, open the project properties, select **Build Properties**, and on the Build Paths tab page click **Generate**.
2. Optionally, specify include directives to ignore.
3. Click **Next**.
4. On the Resolve Include directives dialog, if there are unresolved directives, you might click, for instance, **Resolve All**. The resulting include paths will be stored in the build properties of the selected project.

   You may also choose **Resolve**, **Add**, **Substitute**, and so on.
5. Click **Next**.

The results are stored in the build properties of the project.

Setting Build Properties for User-Defined Projects

For user defined projects, you must edit makefiles.

You may need to edit the build command and default build rules in the build properties. For that purpose, select the project in the Project Explorer view and navigate to the **Build Properties** page.

Then:

- Set a proper build command in the **Build command** field on the Build Support tab page, if necessary.
- Set a **Build Project rule** and **Build folder rule** on the Build Targets tab page, if you use any other than the all rule in your makefile.

Build output is analyzed to determine symbols and include paths that the indexer should use to parse the files of the project. Only files parsed during a previous build run are parsed.

NOTE: If no files have ever been built for a project , all files are parsed, guaranteeing that you end up with an index, source analysis, and navigation capabilities.

10.5.3 Setting Up Paths and Symbols

Usually you do not need to set up the indexer manually by defining additional symbols and include paths. The indexer derives the necessary settings automatically from the build properties of managed projects, or from the build output of user-defined projects.

In addition, for user-defined projects, include paths are automatically calculated after creating a project, in order to have a valid setup even before the project is built. (Files and folders are automatically considered only if they have been added or linked to the project at project creation time.)

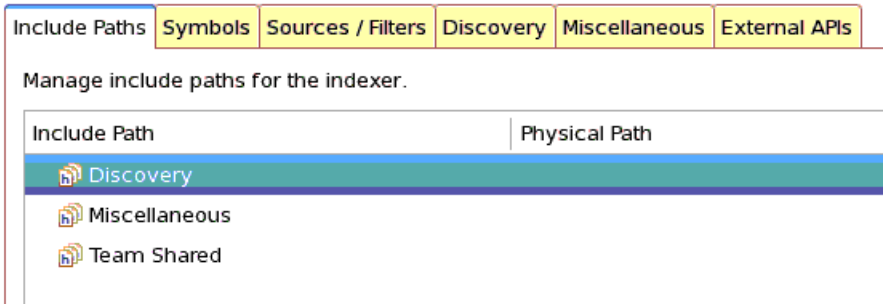
At any time later, you can let Workbench detect include paths for a project, using the Include Search Path (ISP) wizard. In order to do so, follow the directions in the following subsections for both managed and user-defined projects.

NOTE: The ISP wizard for managed projects (via **Build options > Generate include search paths**) is only for managed projects. It stores its results in the build properties of the corresponding project. However, the ISP in the Include Paths tab page (see *Managing Include Paths for the Indexer (Include Paths Tab)*, p.181) is available for both managed and user-defined projects, and it stores its results in the team-shared include paths of the corresponding project. (This wizard is always available when you select the **Team Shared** node in the Include Paths tab page.)

To access the ISP wizard:

1. Right-click the project.
2. Choose **Properties > C/C++ General > Paths and Symbols**,.
3. Notice the tabs in the resulting Paths and Symbols page.

The following subsections describe how to use the pages brought up by these tabs: Include Paths, Symbols, Sources / Filters, Discovery, Miscellaneous, and External APIs, as shown on the following figure.



Managing Include Paths for the Indexer (Include Paths Tab)

The Include Paths page lets you assign or specify include paths in these categories:

- Discovery — Enable build-output analysis for user-defined projects. Highlight Discovery and click Edit to branch to the Discovery tab (see *Setting Up a Build-Driven Index (Discovery Tab)*, p.184).
- Miscellaneous — Configure project-specific options and compiler name and location. Highligh Miscellaneous and click Edit to branch to the Miscellaneous tab (see *Specifying User-Private Paths and Symbols (Miscellaneous Tab)*, p.185).
- Team Shared — A set of include paths shared automatically in a team.

Add Folder
Adds an include path marked by a folder icon

Add File
Adds an include file marked by a header file icon

Generate
Detects required include paths for unresolved include directives by analyzing the source files of the current project.

Edit
Changes an entry, in effect, replacing it

Substitute
Allows replacing of path fragments by placeholders (for instance, build macros, Wind River environment variables such as WIND_Home, project location variables, and so on)

Share
Copies selected include paths and files from any section to the Team Shared sectionan include path marked by the folder icon

Copy
Copies an include path or file

Remove
Removes an include path or file

Using Team Shared Paths

Choose the Team Shared set of include paths, and then one of the button actions in the right column: Add Folder, Add File, Generate, Substitute, or Remove.

For example:

1. Select an item or items in the **Team Shared** section:
2. Click **Generate**.

   The resulting settings are stored with others in the **.cproject** file under the project folder. You can now check this in with your project in your vesion control system.

NOTE: You must *not* modify the **.cproject** file manually because it is an essential part of the project.

3. Optionally, specify include directives to ignore.
4. On the first wizard page, select options for anlyzing the include statements of your sources.
5. Click **Next**.
6. On the Resolve Include directives dialog, if there are unresolved directives, you might click, for instance, **Resolve All**. The resulting include paths will be stored in the build properties of the selected project.

   You may also choose **Resolve**, **Add**, **Substitute**, and so on.
7. Click **Finish**.

Any include paths resulting from Steps 6 and 7 will be stored as team-shared settings. You can click **Copy** on the Include Paths page, and paste the resulting paths into your makefile, if necessary. You can also chose **Add**, **Remove**, **Edit**, and so on, to edit the include paths. (**Share** is only possible for selections not including team-shared paths.)

If you encounter troubles with source navigation tools for certain files, see *A.4 Fixing Indexer Issues*, p.341.

To share an index, see *10.5.5 Sharing Source Analysis Data with a Team*, p.187.

Configuring Indexing of Symbols (Symbols Tab)

In the Symbols tab page of the Paths and Symbols dialog, you can choose how to manage symbols that are in a Team Shared, Miscellaneous, or Build section.

For Team Shared symbols, you can use the following buttons:

- **Add**, **Remove** — to add or remove team-shared symbols
- **Edit** — to modify such symbols or to navigate to the properties page where they can be defined
- **Share** — to copy include paths of other sources into the team-shared section.

Values need not be quoted or escaped. They are passed to the indexer as they are. For example, click **Add**, set **MACRO5** as name, and **'c'** as value. This corresponds to **`#define MACRO5 'c'`**.

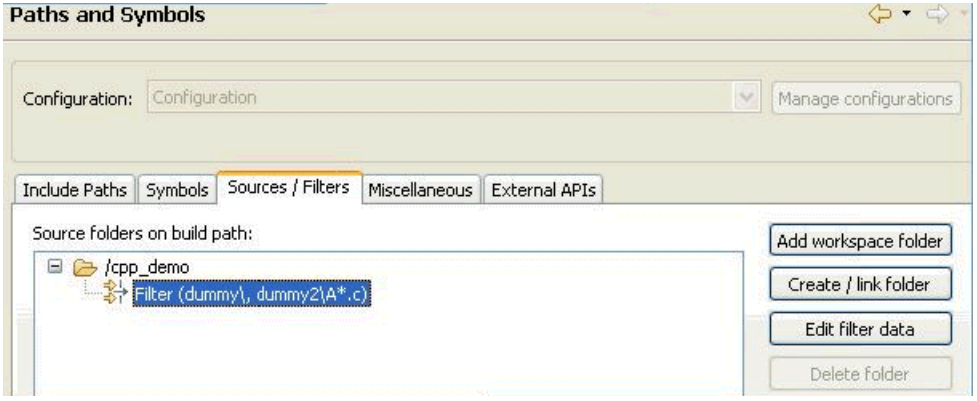
Configuring Sources and Exclusion Filters (Sources / Filters Tab)

In the Sources/Filters tab page of the Paths and Symbols dialog, you can specify exclusion filters for the project.

Excluded files and folders are not parsed, even if the **Index all files** option is set for the project (see *10.5.1 Setting Indexer Preferences*, p.176).

The root nodes are source folders. Per default, there is only one single source folder—the project root folder. You should never replace this by another source folder. Additional source folders, however, are allowed.

Figure 10-1 **Sample Exclusion Filter**



In order to specify an exclusion filter, select a source folder, and click **Edit filter data**. You can then add files and folders to be excluded. Wildcards are supported.

In the example in the preceding figure, the filter entry **dummy/** causes the **dummy** sub-folder under the cpp_demo project to be excluded. The **dummy2/A*.c** part excludes all files of the **dummy2** sub-folder starting with **A** and having a **.c** extension. If the filter were ****/dummy3**, it would exclude all **dummy3** sub-folders of the cpp_demo project or of any sub-folder of **cpp_demo**.

If you need to include a sub-folder **dummy4** of an excluded folder **dummy3**, then simply add **dummy4** as source folder by using the **Add workspace** folder button. **dummy4** will then be treated as exception to the **dummy3** exclusion filter.

Setting Up a Build-Driven Index (Discovery Tab)

The Discovery tab page of the Paths and Symbols dialog provides options for controlling the build-output analysis for user-defined projects. (The page is only visible for this kind of project.)

Use Discovery to enable build-output analysis where you want to set up the project according to your build settings. You cannot modify the results. Click **Clear** to reset the discovered include paths, symbols, and build scope, if necessary.

When the **Enable analysis of build output from console and build-output file** option is enabled, include paths, symbols, and the build-scope (the set of built files) are gathered during a build. The results are used to set up the indexer correctly. Although user-defined projects already come with a set of include paths found during project creation (by analyzing source code), the results delivered by build-output analysis are much more accurate, since the results reflect the actual build settings. The impact on the build-performance is minimal, so you should leave this option enabled. Disable this option only if the results are not sufficient.

The **Discovery statistics** section reports the number of discovered include paths, symbols and source files, and helps you to assess the correctness of the results. In addition, include paths and symbols are presented on the Include Paths and Symbols tab pages.

Use the **Analyze build-output file** section to perform build-output analysis on a build -output file. If a build-output file already exists for your project, then this is the fastest way to set up the indexer correctly, without the need to build the whole project.

Specifying User-Private Paths and Symbols (Miscellaneous Tab)

The Miscellaneous tab page of the Paths and Symbols dialog specifies user-private include paths and symbols.

The edit field takes most common preprocessor options, such as -I, -include, -D, and so on. Include paths (**-I** *includePath*) and include files (**-include** *headerFile*) can contain placeholders, such as $(WIND_BASE). Use **-include** if you want the indexer to treat every source file in the project as if it were including the given header file. This does not have any affect on any build settings.

Symbol options (**-D***name* or **-D***name***=***value*) do not need any quoting for integer values. For instance, -DMACRO1 or -DMACRO1=1 corresponds to #define MACRO1 and #define MACRO1 1. One level of quotes is removed according to compiler command line options.

In order to specify character values for symbols, use single quotes surrounded by double quotes (for instance, -DMACRO2= "'c'"). For string values use double quotes surrounded by single quotes (for instance, -D'MACRO3="Text"' or -DMACRO3='"Text")'.

The compiler inspection field is disabled for managed projects. For user-defined projects, it is filled automatically with the compiler name or path used for building the project (build-output analysis must be enabled). The compiler is used to detect built-in include paths and symbols. If the field is empty, no compiler-internal

include paths and symbols are passed to the indexer. Change the compiler name only if necessary.

NOTE: Miscellaneous settings are lost when the project is removed from the workspace.

Specifying External APIs (External APIs Tab)

Check or un-check **Enable project-type specific external API**.

An *external API* is an index holding a pre-parsed platform API for a project type. Most managed project types come with an external API package.

The external APIs are used for code completion and parameter hinting (hover info) in the source editor. In addition, they can be used to find out where specific platform API functions are declared. For example, when you use **Open Element** or **C/C++ Search** to search for a function named **printf**, you get the locations of the function declarations with this name, even if the project itself does not contain the corresponding header files.

In other words, this allows you to find out which header files to include to build your sources successfully.

Turning off the external APIs for a project is useful, if, for example, you are working on the source files that are actually the basis for the external APIs. This avoids having two different indexes for the source files in the project (one for the sources in the project, and one for the corresponding external API that might reference outdated versions of the same source files).

10.5.4 Updating a Project's Index

Changing include paths or symbols, for example, via the **Build Properties** page or the **Paths and Symbols** property page, has an immediate effect only on parsing modified and newly added files. For performance reasons, the index is not rebuilt automatically after such changes. You have three different options to update the index manually in order to get more accurate source navigation:

- **Rebuild** — Right-click the project in the Project Explorer and select **Index > Rebuild** in order to clear the index and re-parse all files in the current build-scope (for build-driven setup) or all files in the project (no build target, or the **Index All Files** option is enabled for a project, or build-output analysis is disabled or did not return any results).

The current include paths, symbols, and build scope will be used to re-parse all files, thus source navigation gives the most accurate results after performing a full rebuild.

- **Update with Modified Files** — Right-click the project in the Project Explorer and select **Index > Update with Modified Files** in order to parse modified and newly added files.
- **Freshen All Files** — Right-click the project in the Project Explorer and select **Index > Freshen All Files** in order to re-parse previously parsed files with the current include path and symbol settings.

For either option, the current include path and symbol settings defined for the project are used when parsing files.

10.5.5 Sharing Source Analysis Data with a Team

You can create a project index to be shared with project team members. Such a team-shared index holds a snapshot of source-analysis data retrieved by parsing the project.

When a project is imported, this significantly reduces the initial parsing time for a project, because only modified files are parsed (as compared with parsing all when the index was created).

After the project is imported, only the pre-filled project index is used — the team-shared index has no further relevance.

In order to create and share an index:

1. Right-click a project in the Project Explorer
2. Select **Export**.
3. In the Export dialog, select **C/C++ > Team Shared Index**.
4. Click **Next**.
5. Mark other open projects if you want to export their indexes.
6. Select an export destination relative to the project root folder.
7. Use **Insert variable** if you need to store the index outside of the project, in order to avoid absolute paths. Make sure that the export destination is available by all team-members (access to an NFS or Windows/Samba share).
8. Check the created **.settings** project subdirectory including all files in this folder into your version control system to share it with other team members.

The team-shared index will be loaded automatically whenever the project is opened. No user interaction is required.

The file **.settings/org.eclipse.cdt.core.prefs** under your project folder holds the location of the team-shared index (slot **indexer/indexImportLocation**). Do not edit this entry, unless the location of the index has moved outside the project, and you are not willing to export the index again.

11 Building Projects

11.1 Introduction 189

11.2 Configuring Managed Builds 191

11.3 Configuring User-Defined Builds 195

11.4 Accessing Build Properties 196

11.5 Makefiles 197

11.1 Introduction

Workbench offers several levels of build support or build types

These build types are available from **Window > Preferences > Wind River > Build > Build Properties**, where you can set Managed, Disabled, or User-defined builds, depending on whether your projects are Customer-Specific Linux, or Native Application, or User-Defined projects.

You can also set them from a project's **Project > Properties > Build Properties**, again, depending on the type of project.

These build types are as follows:

Managed Build

Workbench controls all phases of the build. Managed build support is available for all project types except user-defined projects.

NOTE: *Managed build* refers to the type of build that was called *flexible managed build* in previous releases of Workbench. What used to be called *standard build* or *standard managed build* has been deprecated, but you can still create one by choosing **Preferences > Wind River > Build** and checking the box for **Enable deprecated standard managed build**, if your build structure is similar to the file system structure. *You cannot change a project's build type*.

User-Defined build

With User-defined builds, you are responsible for setting up and maintaining your own build system and Makefiles, but Workbench does provide minimal build support.

- You can configure the build command used to launch your build utility, so you can start builds from the Workbench GUI.
- You can create build targets in the Project Explorer that reflect rules in your makefiles, so you can select and build any of your make rules directly from the Project Explorer.

Workbench displays build output in the Build Console.

Disabled build

If you select Disabled build for a project or folder, Workbench provides no build support at all. This is useful for projects or folders that contain, for example, only header or documentation files that do not need to be built.

Disabling the build for such folders or projects improves performance both during makefile generation as well as during the build run itself.

11.2 Configuring Managed Builds

Once your project is created, you will see a **Build Targets** node inside it. Your project contains the usual project files, but you must create a build target manually.

Adding Build Targets to Managed Builds

To add a build target to your project:

1. Right-click the **Build Targets** node and select **New Build Target**.

   The **New Build Target** dialog appears. By default the **Build target name** and **Binary output name** are the same as the project name, but if you are going to create multiple build targets, enter more descriptive names.

NOTE: Your build targets must have unique names, but you can use the same Binary output name for each one. This allows you to deliver an output file with the same name in multiple configurations. Workbench adds a build tool-appropriate file extension to the name you type, so do not include the file extension in this field.

2. Choose the appropriate **Build tool** for your project, then click **Next**. The **Edit Content** dialog appears.
3. To display files, folders, and other build targets from outside your current project, select **Show all projects**.

   If you have created a **Working Set**, you can restrict the display by selecting it from the pull-down list.
4. You can add contents to your build target in several ways:
   a. You can select specific files, folders, projects, or other build targets in the left column and click **Add**. What you can add depends on the build tool you use; for example, you cannot add an executable build target to another build target.

      When choosing folders or projects, they can be added “flat” or with recursive content.

      – Clicking **Add** creates a “flat” structure, meaning that Workbench adds the exact items you choose and skips any subfolders and files.

      – Clicking **Add Recursive** creates a structure that includes subfolders and files.

NOTE: Adding linked resources to a build target may cause problems within a team if the linked resources are added using an absolute path instead of a variable.

To define a path variable, select **Window > Preferences > General > Workspace > Linked Resources**, click **New**, then enter a variable name and location.

b. You can create a virtual folder within your build target by clicking **Add Virtual Folder**, typing a descriptive name in the dialog, and clicking **OK**. Virtual folders allow you to group objects within the build target so you can apply the same build settings to them; they also provide a way to add files with the same name from different locations.

   – To add contents to your virtual folder, right-click it in the Project Explorer and select **Edit Content**.

   – Select content as described in step a above, and click **Finish**.

5. To adjust the order of the build target contents, select items in the right column and click **Up**, **Down**, or **Remove**.

NOTE: Folders appear in the specified place in the list, but the files within them are added alphabetically.

6. When you have configured your build target, click **Finish**. It appears in the Project Explorer under the **Build Targets** node of your project.

Modifying Build Targets

There are several ways to modify your build target once it has been created.

Editing Content

To add additional items, adjust the order, or make any other changes to your build target, right-click it in the Project Explorer and select **Edit Content**. The **Edit Content** dialog appears, with the build target content displayed in the right column. Adjust the contents as necessary, then click **Finish**.

Renaming Build Targets and Virtual Folders

To rename your build target or virtual folder, select it in the Project Explorer, press **F2**, and type a new name.

Copying Build Targets

To copy a build target, right-click the build target and select **Copy**, then right-click the destination project's **Build Targets** node and select **Paste** (if you are pasting back into the original project, type a unique name for the new build target).

This is useful for setting up the same build targets in multiple projects with different project types (for example, a library for a native application and a downloadable kernel module will have the same contents but different flags).

NOTE: The build target and its contents are copied, but any overridden attributes are not.

Removing Content

To remove an item from the build target, right-click it in the Project Explorer and select **Remove from Build Target**, or just select it and press **Delete**.

Depending on the item you selected, the menu item may change to **Exclude from Build Target** if the item cannot be deleted (for example, recursive content cannot be deleted). Pressing **Delete** also reinstates an item by removing the exclusion.

Excluding Content

To exclude a specific item from the build target that was included recursively, right-click it in the Project Explorer and select **Exclude from Build Target**.

You can also use regular expressions to exclude groups of items.

1. To add a pattern to the excludes list, right-click a folder in the build target, then select **Properties**, then select the **Excludes** tab.
2. Click **Add File** to define a pattern to exclude specific files or file types. For example, type ***_test.c** to exclude any file named *filename*_**test.c**.

   You can include additional parts of the path to better define the file you want to exclude; for example, type **lib/standard_test.c** to exclude that specific file.
3. Click **Add Folder** to define a pattern to exclude folders within specific folders. For example, type ***/lib/*_test.c** to exclude any file located in a folder named **lib** and named *filename*_**test.c**.

Leveling Attributes

The leveling chain for managed build projects is shown below.

Project > Target > Folder > File
Project > Target > Folder > Subfolder > File
Project > Target > Virtual folder > File
Project > Target > Virtual folder > Folder >
Project > Target > File

The folder level here is related to folders underneath a build target, as described in *Adding Build Targets to Managed Builds*, p.191.

You can now configure the build target with specific settings for all build tools on a build target level (for example, you can set compiler options for the source files related to that build target).

Understanding Managed Build Output

Workbench does not create build redirection directories for each folder, as the objects might be built differently when building them for specific targets. Instead, Workbench creates a build-specific redirection directory, which you can configure on the **Build Properties** > **Build Paths** tab, underneath the project root directory.

In this redirection directory there is a directory for each build-target, and inside those are directories named **Debug** or **NonDebug** depending on the debug mode you chose for the build. Workbench generates the output files according to the structure you defined in the build target, and deposits them in the debug-mode directory.

In general, the build output is structured like this:

Project directory
Project dir/build specific redirection dir
Project dir/build specific redirection dir/target dir
Project dir/build specific redirection dir/target dir/debug mode dir
Project dir/build specific redirection dir/target dir/debug mode dir/binary output file of the build target

All objects belonging to the build target are stored within an additional **Objects** subfolder:

Project dir/build specific redirection dir/target dir/debug mode dir/Objects/structure of object files

Example Build Target and Build Output Structure

To understand how the build target structure influences the build output, below is an example of a project source tree.

```
proj1/
proj1/a.c
proj1/b.c
proj1/folder1/c.c
proj1/folder1/d.c
```

Target1 contains these two items:

```
a.c
folder1/*.c
```

Target2 contains these two items:

```
b.c
d.c
```

Configuring the project to use **spec1** as the active build spec, naming the redirection directory **spec1**, and turning debug-mode **on** produces the output structure seen below.

```
proj1/spec1/Target1/Debug/Target1.out
proj1/spec1/Target1/Debug/Objects/a.o
proj1/spec1/Target1/Debug/Objects/folder1/c.o
proj1/spec1/Target1/Debug/Objects/folder1/d.o

proj1/spec1/Target2/Debug/Target2.out
proj1/spec1/Target2/Debug/Objects/b.o
proj1/spec1/Target2/Debug/Objects/d.o
```


11.3 Configuring User-Defined Builds

When you create a User-Defined project, you can configure the build command, make rules, build target name, and build tool (for more information, see *7. Creating User-Defined Projects*). To create the build target, right-click your project in the Project Explorer and select **Build Project**.

To update the build settings, right-click your project in the Project Explorer and select **Properties**, then select **Build Properties**.

For more information about the settings described on the build properties tabs, open the build properties dialog and press the help key for your host.

11.4 Accessing Build Properties

There are two ways to set build properties: in the Workbench preferences, to be automatically applied to all new projects of a specific type, and manually, on an individual project, folder, or file basis. The properties displayed will differ depending on the type of node and the type of project you selected, as well as the type of build associated with the project.

For details, see *Wind River Workbench User Interface Reference: Build Properties*.

11.4.1 Workbench Global Build Properties

To access global build properties, select **Window > Preferences** and choose the **Build Properties** node.

This node allows you to select a project type, then set default build properties to be applied to all new projects of that type.

11.4.2 Project-specific Build Properties

To access build properties from the Project Explorer, right-click a project and select **Properties**. In the Properties dialog, select the **Build Properties** node.

The project-specific Build Properties node has tabs that are practically identical to the ones in the Workbench preferences, but these settings apply to an existing project that is selected in the Project Explorer.

11.4.3 Folder, File, and Build Target Properties

Folders, files, and build-targets inherit (reference) project build properties where these are appropriate and applicable. However, these properties can be overridden at the folder/file level. Inherited properties are displayed in blue typeface, overridden properties are displayed in black typeface.

Overridden settings are maintained in the **.wrproject** file. This file should therefore also be version controlled. Note that you can revert to the inherited settings by clicking the eraser icon next to a field.

11.4.4 Multiple Target Operating Systems and Versions

If you installed Workbench for multiple target operating systems and/or versions, you can set a default target operating system/version for new projects in the Workbench Preferences, under **General > Target Operating Systems**.

For existing projects, you can verify the target operating system (version) by right-clicking the project in the Project Explorer, then selecting **Properties**, then **Project Info**.

NOTE: In most cases, it is *not* possible to successfully migrate a project from one target operating system or version to another simply by switching the selected **Target Operating System and Version**.

In the Project Explorer (and elsewhere), the target operating system and version are displayed next to the project name by default. You can toggle the display of this information in the **Preferences > General > Appearance > Label Decorations**, using the **Project Target Operating Systems** check box.

If you have multiple versions of the same operating system installed, the New Project wizard allows you to select which version to use when creating a new project.

11.5 Makefiles

The build system uses the build property settings to generate a self-contained makefile named **Makefile**. For managed builds, only one Makefile is created per build spec.

By default makefiles are stored in project directories; if you specified an absolute **Redirection Root Directory** (for details, open the Build Paths tab and press the help key for your host), they are stored there, in subdirectories that match the project directory names.

The generated makefile is based on a template makefile named **.wrmakefile** that is copied over at project creation time. If you want to use custom make rules, enter these in **.wrmakefile**, *not* in **Makefile**, because **Makefile** is regenerated for each build. The template makefile, **.wrmakefile**, references the generated macros in the

placeholder **%IDE_GENERATED%**, so you can add custom rules either before or after this placeholder. You can also add ***.makefile** files to the project directory.

For other ways of setting custom rules, see *12.6 User-Defined Build Targets in the Project Explorer*, p.210.

NOTE: If you configure your project for a remote build, the generated Makefile contains paths for remote locations rather than local ones. For more information about remote builds, see *12.7 Developing on Remote Hosts*, p.213.

11.5.1 Derived File Build Support

The Yacc Example

Workbench provides a sample project, **yacc_example**, that includes a makefile extension showing how you can implement derived file build support. It is based on **yacc** (Yet Another Compiler Compiler) which is not contained in the Workbench installation. To actually build the example, you need to have **yacc** or a compatible tool (like GNU's **bison**) installed on your system, and you should have extensive knowledge about make.

The makefile, **yacc.makefile**, demonstrates how a yacc compiler can be integrated with the managed build and contains information on how this works.

1. Create the example project by selecting **New > Project > Example > Native Sample Project > Yacc Demonstration Program**.
2. Right-click the **yacc_example** project folder, then select **New > Build Target**. The New Build Target dialog appears.
3. In the **Build target name** field, type **pre_build**.
4. From the **Build tool** drop-down list, select **(User-defined)**, then click **Finish** to create the build target.
5. In the Project Explorer, right-click **pre_build** and select **Build Target**. This will use the makefile extension **yacc.makefile** to compile the yacc source file to the corresponding C and header files. The build output appears in the Build Console.

NOTE: Execute this build step prior to the project build, or the files generated by **yacc** will not be used by the managed build, which generates the corresponding makefile before the build is started and before all files that are part of the project at this time are taken into account.

6. When the build is finished, right-click the **yacc_example** folder and select **Build Project**.

Additional information on how you can extend the managed build is located in **yacc.makefile**. It makes use of the extensions provided in the makefile template **.wrmakefile**, which can also be adapted to specific needs.

General Approach

To implement derived file support for your own project, create a project-specific makefile called *name_of_your_choice***.makefile**. This file will automatically be used by the managed build and its make-rules will be executed on builds.

It is possible to include multiple ***.makefile** files in the project, but they are included in alphabetical order. So if multiple build steps must be done in a specific order, it is suggested that you use one ***.makefile** and specify the order of the tools to be called using appropriate make rules.

For example:

1. Execute a **lex** compiler.
2. Execute a **yacc** compiler (depending on **lex** output).
3. Execute a SQL C tool (depending on the **yacc** output).

Solution: (using the **generate_sources** make rule)

```
generate_sources :: do_lex do_yacc do_sql
do_lex:
    @...

do_yacc:
    @...

do_sql:
    @...
```

or

```
generate_sources :: $(LEX_GENERATED_SOURCES) $(YACC_GENERATED_SOURCES)
$(SQL_GENERATED_SOURCES)
```

Add appropriate rules like those shown in the file **yacc.makefile**.

12

Building: Use Cases

12.1 Introduction 201

12.2 Adding Compiler Flags 202

12.3 Building Applications for Different Target Architectures 204

12.4 Creating Library Build Targets for Testing and Release 204

12.5 Implementing Architecture-Specific Functions 207

12.6 User-Defined Build Targets in the Project Explorer 210

12.7 Developing on Remote Hosts 213

12.1 Introduction

This chapter suggests some of the ways you can go about completing various build-specific tasks in Wind River Workbench.

12.2 Adding Compiler Flags

You may know the exact compiler flag you want to add, for example **-w**, and use the GUI to help you put it in the right place as shown in *Adding a Compiler Flag by Hand*, p.202. You can also use the GUI to help you specify the correct compiler flag for what you want to do, as shown in *Adding a Compiler Flag with GUI Assistance*, p.203.

NOTE: This section describes how to add and edit the compiler flags on specific projects. However, for Wind River Linux projects, global changes are made in board templates so that the settings can be used by both the command line build system and Workbench, as described in *Wind River Linux Platforms User's Guide*.

Adding a Compiler Flag by Hand

If, for example, you are familiar with the GNU compiler command line and you just want to know *where* to enter the **-w** option:

1. In the Project Explorer, right-click an application project and select **Properties**.
2. In the Properties dialog box, select the **Build Properties** node.
3. In the **Build Properties** node, select the **Build Tools** tab.
4. In the **Build Tools** tab:
   a. Set the **Build tool** to **C-compiler**
   b. Set the **Active build spec** to, for example, **PENTIUM-gnu-native**.
   c. In the field next to the **Tool Flags** button, append a space and **-w**.

      The contents of this, the **Tool Flags** field you have just modified, is expanded to the **%ToolFlags%** placeholder you see in the **Command** field above it. Because you entered the **-w** in the **Tool Flags** field, rather than the **Debug** or **Non Debug** mode fields, warnings will always be suppressed, rather than only in either **Debug** or **Non Debug** mode**.**

Adding a Compiler Flag with GUI Assistance

If you are not familiar with the specific command line tool options that you want to use, the GUI may be able to help. For example:

1. In the Project Explorer, right-click an application project, and select **Properties**.
2. Click **Build Properties** and select the **Build Tools** tab.
3. In the **Build Tools** tab:
   a. Set the **Build tool** to **C-compiler**
   b. Set the **Active build spec** to, for example, **PENTIUM-gnu-native**.
   c. We assumed you are unfamiliar with the GNU compiler options so, to open the GNU Compiler Options dialog box, click the **Tool Flags** button.
   d. In the GNU Compiler Options dialog box, click your way down the navigation tree at the left of the dialog box and take a look at the available options.
   e. When you get to the **Compilation > Diagnostics** node, select the check box labelled **Suppress all compiler warnings**.

      Notice that **-w** now appears in the list of command line options at the right of the dialog box.
   f. Click **OK**.
4. Back in the **Build Tools** node of the Properties dialog box, you will see that the **-w** option you selected now appears in the field next to the **Tool Flags** button.

   The contents of this, the **Tool Flags** field, is expanded to the **%ToolFlags%** placeholder you see in the **Command** field above it.

12.3 Building Applications for Different Target Architectures

The target nodes under projects in the Project Explorer display, in blue, the name of the currently active build spec. You may want to switch build specs to build projects for different architectures.

If, for example, you want to build an application for testing on the localhost, and then build the same project to run on a real board, you would simply switch build specs as follows:

1. Right-click the project node and select **Build Options > Set Active Build Spec**.
2. In the dialog box that appears, select the build spec you want to change to and specify whether or not you want debug information.

   When you close the dialog box, you will notice that the label of the target node has changed. If you selected debug mode in the dialog box, the build spec name is suffixed with **_DEBUG**.
3. Build the project for the new architecture.

NOTE: To select the Active Build Spec directly from the Project Explorer, click the checkmark icon in the Project Explorer toolbar.

12.4 Creating Library Build Targets for Testing and Release

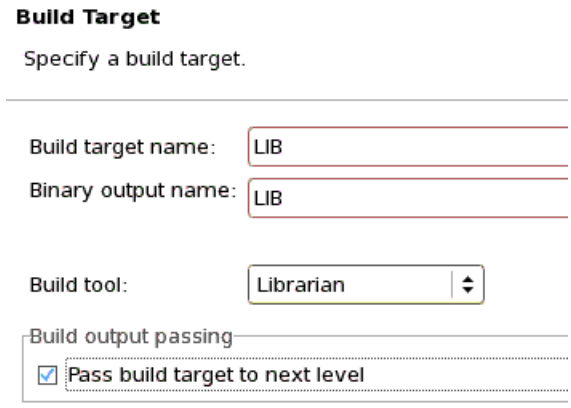
Assume you have a library project that consists of the files **source1.c**, **source2.c**, and **test.c**. The file **test.c** implements a **main( )** function and is used as an executable for testing the library, therefore it is not to be included in the library.

This implies the use of different build targets that are built with different build tools, as described below. The application project type is appropriate because you need to use both the **Linker** and the **Librarian** build tools.

1. Create an application project to hold all the files mentioned above:
   a. Select **New > Wind River Linux Application Project**.
   b. Name the project **LIB** (for example).

c. Click **Next** until you reach the **Build Target** dialog.

d. Change **Build tool** to **Librarian** and check **Pass build target to next level**.

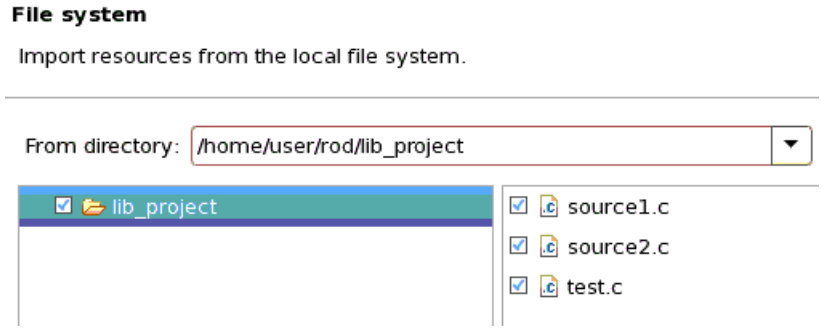


e. Click **Finish**.

2. Import the library files into the newly created LIB project:

a. Right-click the project, select **Import > File System**, and click **Next**.

b. Browse to the directory containing the library files, then, in the left pane, select the check box in front of the directory name.

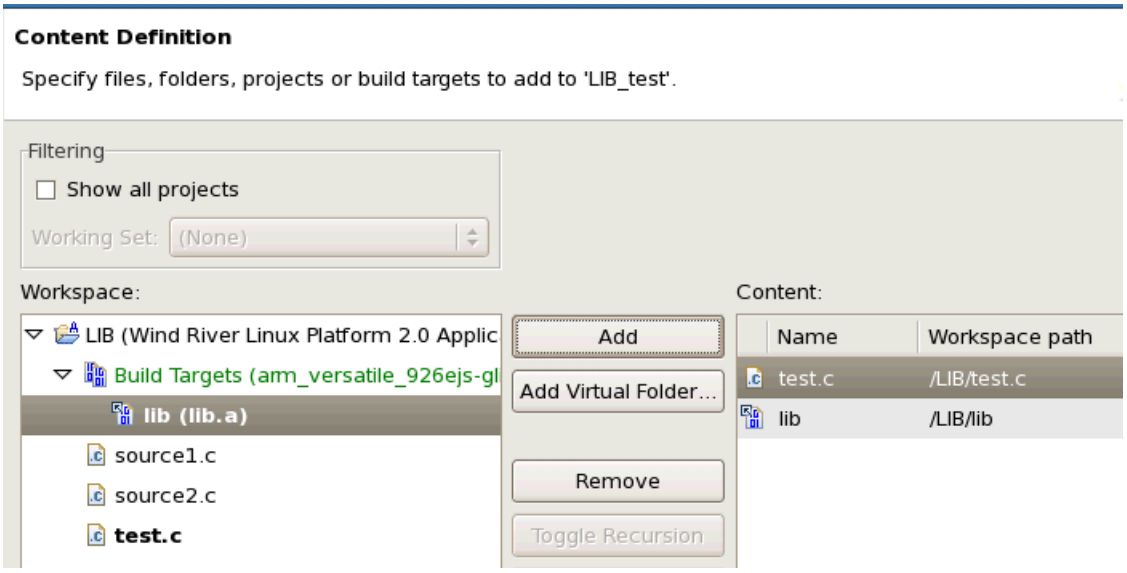


All the files in the directory are imported by default.

c. Click **Finish**.

3. Create a target for the test version of the library:
   a. Right-click the green **Build Targets** node under the **LIB** project, and choose **New Build Target**.
   b. Name the new target **LIB_test** (in the **Build target name field**).
   c. Set the **Build tool** to **C-Linker**.
   d. Clear the **Use default content** check box.
   e. Click **Next**.
4. Specify the contents of the test library:
   a. Right-click the green **LIB_test** build target and choose **Edit Content**.

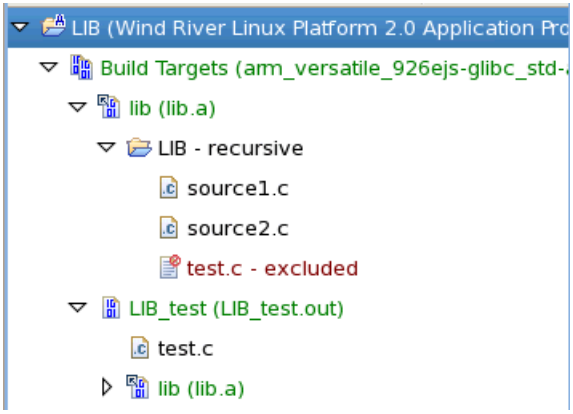
      The Content Definition dialog opens.
   b. Select **source1.c** and click **Add**. Do the same for **source2.c**.



   c. Click **Finish**.
5. Exclude **test.c** from your **LIB.a** library:
   a. Expand the **LIB - recursive** build target to reveal the filenames.
   b. Right-click **test.c** and choose **Exclude from Build Target**.

Your finished LIB project should now contain two build-target nodes:

- **Lib.a**, built from **source1.c** and **source2.c**.
- **LIB_test**, built from **lib.a** and **test.c**.



When you build **LIB_test**, **lib.a** automatically gets built if it is out of date, because of the reference to **lib.a** in the **LIB_test** build target.

12.5 Implementing Architecture-Specific Functions

This section describes architecture-specific implementations at the folder and build-target levels.

To allow architecture-specific implementation of functions within the same project, you can enable and disable build specs:

- At project, folder, build-target, and file levels for flexible managed builds
- At the project and folder levels for standard managed builds (deprecated but available if you check **Enable deprecated standard managed build** under **Preferences > Wind River > Build**)

Consider a simplified flexible managed build project with two subfolders, **arch1** and **arch2**, that each use code-specific differences for different target architectures. You can set up a project to build a software target that requires the implementation

of a function specific to different target boards, where you only need to change the active build spec at the project level.

In this example, suppose that the function **int arch_specific (void)** is declared in **arch.h**, and that:

- **arch1.c** implements **arch_specific (void)** for the **common_pc** build spec (the only build spec enabled for the **arch1** folder)
- **arch2.c** implements **int arch_specific (void)** for **arm_versatile_926ejs** (the only build spec enabled for the **arch2** folder)

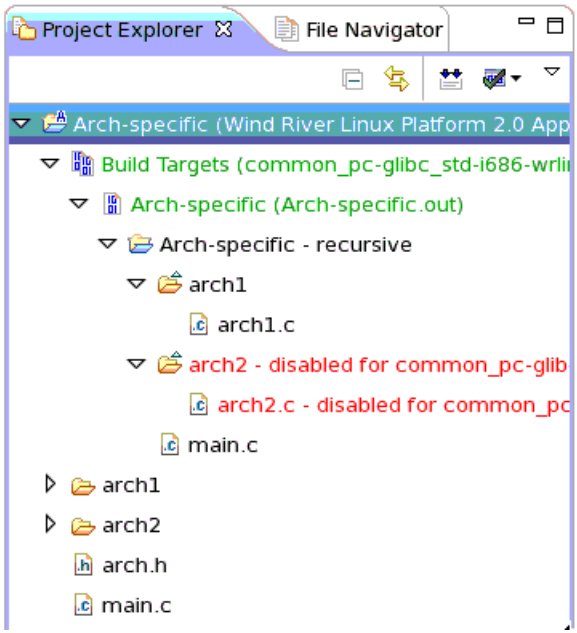
The inner build spec relationships are outlined in Table 12-1.

Table 12-1 **Project Content and Build Spec Configuration of the Structure**

Directories/Folders	Files	Enabled Build Specs
/project	**main.c, arch.h**	**common_pc** and **arm_versatile_926ejs**
/project/arch1	**arch1.c**	**common_pc** only
/project/arch2	**arch2.c**	**arm_versatile_926ejs** only

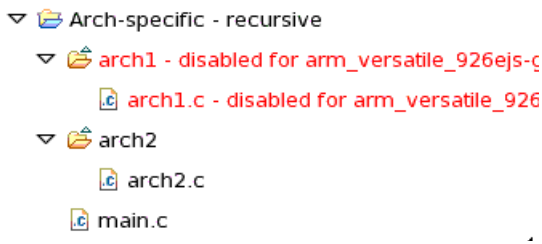
To build the project for the **common_pc** architecture:

1. Highlight the project folder.
2. Click the **Set Active Build Spec** button.
3. Specify **comon_pc as** the active build spec for the project.



The subfolder **arch1** will be built, and its objects will be linked into the project build target. The **arch2** subfolder will not be built, and its objects will not be linked into the project build target because the **comon_pc** build spec is not enabled for **arch2**.

Similarly, if you set **arm_versatile_926ejs** as the active build spec for the project, the **arch2** subfolder will be built, but the **arch1** subfolder will not:



12.6 User-Defined Build Targets in the Project Explorer

In the Project Explorer you can create custom build targets that reflect rules in makefiles. This is especially useful if you have User-Defined projects, which are projects for which Workbench does not manage the build. You might also find this feature useful in other projects as well.

12.6.1 Custom Build Targets in User-Defined Projects

You can define a custom build-target for a rule or rules in your Makefile. To do so:

1. Right-click a project or folder and select **New > Build Target**.
2. In the dialog box that appears, enter the rule(s) you want to create a target for. If you want to execute multiple rules, separate each one with a space.

   The names of the rule(s) you enter must exist in your makefile(s) to be executed when you build your new user-defined target.
3. Set the **Build tool** to **User-defined**.
4. Click **Finish**. The new build-target node appears under the project or folder you selected. The node icon has a superimposed **M** to identify it as a user-defined make rule.

   To execute the rule(s), right-click the new target node and select **Build Target**.

12.6.2 Custom Build Targets in Workbench Managed Projects

First write the make rules you need into the **.wrmakefile** file in the project directory.

NOTE: For Wind River Platform projects, see *12.6.3 Custom Build Targets in Wind River Linux Platform Projects*, p.211.

1. Right-click a project or folder and select **New > Build Target**.
2. In the dialog box that appears, enter the rule name(s) you created in **.wrmakefile**. If you want to execute multiple rules, separate each one with a space.
3. Set the **Build tool** to **User-defined**.

4. Click **Finish**. The new build target node appears under the project or folder you selected. The node icon has a superimposed **M** to identify it as a user-defined rule.

To execute the rule(s), right-click the new target node and select **Build Target**.

12.6.3 Custom Build Targets in Wind River Linux Platform Projects

To create a custom build target for a Wind River Linux Platform project, you edit **Makefile.wr**, not **.wrmakefile**. The contents of **.wrmakefile** (build properties and build targets) should be edited only through the project **Properties > Build Properties** dialogs.

NOTE: This example shows how to add a build target to invoke the uClibc configuration for PCD-based projects. This technique applies to bringing any other Wind River Linux or custom command line feature to Workbench Platform Projects.

First, write the rules you want add to **Makefile.wr**. For example, double-click **Makefile.wr** and add the following at the bottom of the file in the Editor view:

```
uclibc-config :
     xterm - e make -C $(DIST_DIR) uclibc.menuconfig
```

An **xterm** is used because this build rule, **menuconfig**, requires a shell that can support **xterm** commands, which is beyond the capabilities of the **Build Log** view within Workbench.

Note that at the top of **Makefile.wr** are the definitions **DIST_DIR** and **TARGET_DIR**. These provide the redirection to the Wind River Linux Platform project's content directory.

1. Right-click a project or folder and select **New > Build Target**.
2. In the dialog box that appears, enter the rule(s) you want to create a target for. If you want to execute multiple rules, separate each one with a space. For this example, add the build target **uclibc-config**.
3. Set the **Build tool** to **User-defined**.
4. Click **Finish**. The new build target node appears under the project or folder you selected. The node icon has a superimposed **M** to identify it as a user-defined rule.

   To execute the rule(s), right-click the new target node and select **Build Target**.

12.6.4 User Build Arguments

The User Build Arguments drop-down list appears in the Build Console.

At first, the list contains just an empty field. You can enter one or more arguments (such as a rule or rules, or macro re-definitions) that change the execution of any existing make rule, or override any macro, or affect anything else that is understood by make, at every build, regardless of what is being built.

Once you launch a build, the list retains what you enter. You can then type over a list entry. When you perform another build, the list retains your new build arguments as an additional entry.

To enter new arguments:

1. Type one or more arguments into the text field, whether it is blank or already has contents.

   (Use spaces to separate multiple arguments.)

2. Launch a build.

   The build causes the text field's new arguments to be stored in the User Build Arguments list. They are appended to (and thus override) the existing makefile entries. This occurs on the fly at every build.

To use arguments already in the list:

1. Select the appropriate argument or arguments in the list.
2. Launch a build.

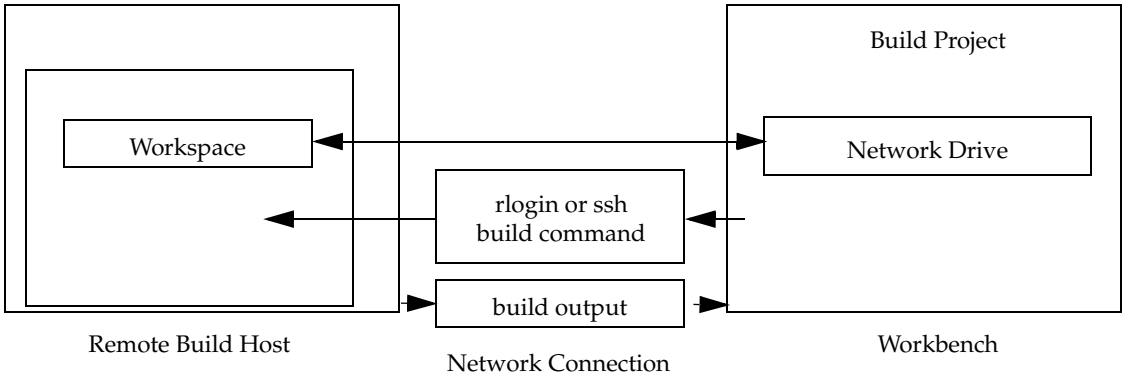
The user build arguments functionality does not provide any value, macro, or shell substitution. For these, set up an intermediate makefile (e.g., **Makefile.wr** for platform projects, or perhaps **Makefile.user**).

NOTE: The current setting of the User Build Arguments field applies to all builds, including those from the **Run Last Build again** button.

12.7 Developing on Remote Hosts

The Workbench *remote build* feature allows you to develop, build and run your applications on a local host running Workbench, using a workspace that is located on a remote host as if it were on a local disk.

Figure 12-1 **Remote Build Feature**



In the case of a managed build, Workbench generates the makefiles on the local machine running Workbench. You map a path from the workspace root location to where generated makefiles will be correctly placed for builds on a remote machine.

When launching the build, Workbench establishes a network connection (**rlogin** or **ssh**) to the build host. The actual build command is executed there by using an intermediate script so you can set up the environment for the build process.

12.7.1 General Requirements

- The Workbench host tools and chain must be installed on the remote machine.
- The workspace root directory has to be accessible from both machines.
- Only Eclipse projects under the workspace root can be remotely built.
- An **rlogin** or **ssh** remote connection to the build machine must be possible.

12.7.2 Remote Build Scenarios

Local Windows, Remote UNIX

The workspace root directory should be located on the remote UNIX host and mapped to a specific network drive on Windows. It may also be possible to locate the root directory on the Windows machine, but then there is the need to mount the Windows disk on the build host. This may lead to problems regarding permissions and performance, so a mapping of the workspace root-directory is definitely needed.

Local UNIX, Remote UNIX

As it is possible to access the workspace root directory on both machines with the equivalent path (automount) it may be possible to skip the path mapping.

Local UNIX, Remote Windows

This scenario is not supported, as you would need to execute the build command on Windows from a UNIX host.

12.7.3 Setting Up a Remote Environment

The **Edit remote command script** button in the Remote Connections dialog box (**Project > Build Options > Remote Connection**) lets you include additional commands and environment variables to set up your environment on the remote machine prior to a build or run.

This button opens **remote_cmd.sh** for editing.

The following version, as an example, has been edited (as indicated by bold font) to update the PATH variable.

```
#!/bin/sh

WORKSPACE_ROOT="%WorkspaceRoot%"
export WORKSPACE_ROOT
DISPLAY=%Display%
export DISPLAY

PATH=/MyTools/gmake_special/bin:$PATH
```

```
export PATH

# translate CR to NL for better build output handling
stty ocrnl 1>/dev/null 2>&1


cd $WORKSPACE_ROOT

cd "$1"
shift 1

exec "$@"
```

You can add any commands you need, but all commands must be in **sh** shell style.

12.7.4 Building Projects Remotely

1. Switch to a workspace that contains existing projects by selecting **File > Switch Workspace**. Type the path to the appropriate workspace, or click **Browse** and navigate to it.
2. In the Project Explorer, right-click a project and select **Build Options > Remote Connection**. The **Remote Connections** dialog box appears.
3. Click **Add** and type a descriptive name for this remote connection. Click **OK**.
4. In the **Connection Settings** fields, add the following information to create a remote connection:

   Connection Type
   Select **Rlogin** or **SSH**.

   Hostname
   The name of the build host (can also be an IP address).

   Username
   The username used to establish the connection (the remote user may differ from the local user).

   Remote Workspace Location
   The root directory of the workspace as seen on the remote host.

   Display (XServer)
   IP address of the machine where the output should be displayed.

   By clicking the **Advanced** button you can also access these fields:

Password request string
A string that will be recognized as a password request to prompt you for the password.

Remember Password during Workbench sessions
A switch to specify whether the password entered should be remembered during the current session. This is useful during a lengthy build/run session.

5. Click **Connect** to connect immediately. Remote connection settings are stored, and are specific to this workspace. They are not accessible from any other workspace.
6. The build is executed on the remote host, with the build output listed in the standard Workbench Build Console. The X Server (IP address listed in the Display field) is used whenever any type of X application is started, either during builds or runs.
7. To return to local development, select **Local Host** from the list of connections, then click **Connect**.

12.7.5 Running Applications Remotely

This section provides information about running native applications only.

Running native applications remotely is quite similar to running applications locally: a **Native Application** launch configuration must be created that defines the executable to be run, as well as remote execution settings for the launch. On the **Remote settings** tab are:

Remote Program
Enter the command that is used to launch the application. This may be useful for command-line applications that could then be launched within an **xterm**, for instance.

Remote Working Directory
This setting is optional, but if a remote working directory is given, it overrides the entry in the **Working Directory** field of the **Arguments** tab.

For remote runs, a new connection similar to the active connection will be established to allow control of Eclipse process handling, as the new remote process will be shown in the Debug view. The Remember password during Workbench sessions feature is very useful here.

Command-line application's output and input is redirected to the standard Eclipse console unless the application is started within an external process that creates a new window (such as **xterm**). The default for remote execution is a remote command like **xterm -e %Application%**, therefore a local X Server (like Exceed or Cygwin X) must be set up and running.

For more information about creating launch configurations, see *17. Launching Programs*.

12.7.6 Rlogin Connection Description

The **rlogin** connection used in the Workbench remote build makes use of the standard **rlogin** protocol and ports. It establishes a connection on port 513 on the remote host, and the local port used must be in the range of 512 to 1023 per **rlogin** protocol convention.

On Windows the **rlogin** connection is implemented directly from within Workbench, so you do not need an existing **rlogin** client. The UNIX implementation is different, because for security reasons the local port (range: 512 to 1023) is restricted to root access, which cannot be granted from within Workbench. Therefore an external **rlogin** process is spawned using the command-line:

`rlogin -l` *username hostname*

rlogin on UNIX platforms makes use of **setUID root** to ensure that the needed root privileges are available.

The standard **rlogin** protocol doesn't support access to **stderr** of the remote connection, to all output is treated as **stdout**. Coloring in the Build Console of Workbench for **stderr** is therefore not available.

NOTE: On Linux the **rlogin** client and server daemon can be switched off per default. So if the machine is used as a Workbench (remote build client) host, the **rlogin** executable must be enabled (or built) and if the machine is acting as build server (remote build host) the **rlogin** daemon must be enabled. Details may be found in the system documentation of the host.

12.7.7 SSH Connection Description

The supported protocol is **SSH2**, and it establishes a connection on port 22 (the default **SSH** port).

Strict host key checking is disabled. Workbench does not use a known hosts file, so host key information is stored in memory, and you are not prompted if the host key changes.

Only password authentication is supported.

12.7.8 Example Using Samba on Remote Linux Host

This section presents a procedure in which you configure Samba on a remote Red Hat Linux host. You then use Workbench on a Windows host to create a project, build it, run the application, and debug it. This example assumes the remote Linux host supports **ssh**.

Configure the Remote Linux Host

The following steps assume an RHEL 4 Linux host. If you are using a different version of Linux you will have to translate the specific commands below for the Samba tools available on your host. The **smb** service should be running on your Linux host (see **System Settings > Server Settings > Services** or **/usr/bin/system-config-services**).

1. Create a workspace directory for a Linux user for which you have log on privileges, or create a new one such as **wbuser**. This is the workspace that you want to export. For example, in your home directory, you could create a subdirectory called **remote_workspace**.

NOTE: This workspace cannot be used remotely by Windows and simultaneously by the local Linux Workbench installation. So either you must not be running Workbench on the Linux host or it must be using a different workspace than the one you specify here.

2. As the root user, start the Samba configuration tool:

```
# /usr/bin/system-config-samba
```

3. Select **Preferences > Samba Users** and click **Add User**. Add the Linux user that has the workspace to be exported.
4. Enter a password for the user. This is the Samba password and can be different than your Windows or Linux password. Click **OK**.
5. Select **Preferences > Server Settings.**

a. In the **Basic** tab enter a workgroup that is accessible from Windows, for example **workgroup**.

b. In the **Security** tab, you should be able to leave the settings as **User** authentication mode, **Yes** for password encryption, and **No guest account**.

c. Click **OK**.

6. Click **Add** to add a Samba share. This is the workspace that you want to export.
   - For **Directory**, enter the full path to the workspace to export, for example **/home/wbuser/remote_workspace**.
   - For **Basic Permissions** check **Read/Write**.
7. Click the **Access** tab and allow access to everyone, or specifically select the user(s) you want to have access. If you want to add specific access for a user that does not appear in this tab, you must add them with **Preferences > Samba Users** as done previously.
8. Click **OK**.
9. Start **usermode-agent**, for example:

   ```
   $ installDir/linux-2.x/usermode-agent/bin/ia/i386/usermode-agent &
   ```


Configure the Windows Host

The following steps assume a Windows XP host. If you are using a different version of Windows there may be some difference in the commands required.

1. Right-click **My Computer** (or select **Tools** in Windows Explorer) and select **Map Network Drive**.
2. Choose a network drive, for example **W:**, and enter your hostname or IP address and the share (workspace), for example:

   ```
   \\remotebox\remote_workspace
   ```
3. Click **Finish**.

NOTE: If you limited access to specific users, you will be prompted to log in with the user name and Samba password.

Configure Workbench

1. Start Workbench on your Windows host and use the exported workspace. If Workbench prompts you for a workspace on startup, enter the drive you mapped to the workspace, for example, **W:**. Alternatively, once Workbench has started, specify the drive from within Workbench by selecting **File > Switch Workspace**.
2. Select **Project > Remote Connection** and click **Add**.
   a. Enter a name you would like to give the remote connection.
   b. Choose the connection type. This example uses **SSH**.
   c. Enter the host name or IP address of the remote Linux host.
   d. Enter the user name of the login account on the remote Linux host.
   e. Enter the full path of the workspace on the remote Linux host, for example, **/home/wbuser/remote_workspace**.
   f. You can ignore the setting for the X server because this example displays output in the Console view. For details on using an X server with a remote host configuration, see *Using an X Server with a Remote Host*, p.222.
   g. Click **Connect**, and click **Yes** when prompted to save your settings.
3. Enter your remote Linux host password when prompted.

NOTE: You are now connecting to the remote Linux host with **ssh** (or **rlogin**) so this must be your Linux password which may be different than your Samba login.

Create an Example Project

1. Right-click in the Project Explorer and select **New > Example > Native Sample Project**. Click **Next**.
2. Select the **Hello World Demonstration Program** and click **Finish**.

   This creates the project. You can now look in the **remote_workspace** directory on your remote Linux host and see that a **hello_world** subdirectory is in it. You can also see the same contents in the Windows drive you have mapped to the share.

3. Click the **Set Active Build Spec and Debug Mode** icon in the Project Explorer and set the active build spec to Linux GNU.
4. Right-click the **hello_world** project and select **Build Project**.The build is performed on the remote Linux host.

Run the Application on the Remote Host

1. Right-click the executable or the **DEBUG** directory and select **Run Native Application**. click the **Remote Settings** tab. In this example we are *not* assuming that the Windows host is running an X server, so delete the string **xterm -e %Application%** in the section under **Remote Program** so that it is empty.
2. If you do not have a Console view in your current Workbench window, select **Window > Show View > Console**. You may want to move it to a place where it is always visible if it is not already.
3. Click **Run**. Build ouput appears in the Build Console view and after a few moments the program output **hello world** appears in the Console view. This output has come from the execution of the program on the remote Linux host.

Debug the Application on the Remote Host

1. Click the **Create a New Target Connection** icon in the Remote Systems view. If you already have a target connection that you want to use, be sure the object path mappings and source path lookup are set correctly as described in this procedure.
2. Select the user mode connection type and click **Next**.
3. Ignore the usermode agent options screen and click **Next**.
4. Enter the remote Linux host name or its IP address and click **Check** to verify it. Click **Next**.
5. Specify your object file path mappings by associating the path to the workspace on the remote Linux host with the drive share under Windows. For example:

   Target path: **/home/wbuser/remote_workspace**

   Host path: **W:**
6. Click **Next** until you can click **Finish** to complete the connection.

7. Select **Run > Debug** and select your new connection. Click **Debug**.

NOTE: If you get a **Source not found** error in the Editor space, click **Edit Source Lookup**, click **Add**, click **Workspace**, and click **OK**. Then expand **Workspace**, select your project folder, and click **OK**. (You can also add this source lookup path to an existing connection by selecting **Run > Debug** and using the **Source** tab.)

You can now proceed to debug the program as usual. Note that program output apears in the Console view.

Using an X Server with a Remote Host

If you have an X server available on your Windows host, such as Exceed or Cygwin X, you can display the remote Linux output in an **xterm** on the Windows host if you are configured as follows:

1. In **Project > Remote Connection**, the **Display (X server)** field is by default set to *IP_address*:**0.0**. This should be acceptable unless you are running multiple X servers on your Windows host.
2. The Windows X server must allow clients from your remote host to display locally. For example, if you are using Cygwin X, enter **xhost +** in an **xterm** window to allow access to all X clients, or **xhost +***IP_address*, using the IP address of your remote Linux host for restricted access.
3. If you are using the **hello_world** sample program, the output will probably display too fast to be seen because the **xterm** closes immediately after displaying the message.
   a. Open the project in Workbench and double-click **helloworld.c** to open it in the Editor.
   b. Add the following **include** statement:

      ```
      #include <unistd.h>
      ```

   c. And add the following line after the **printf** statement:

      ```
      sleep(10);
      ```

   d. Save **helloworld.c**.
4. Right-click the executable or the **DEBUG** directory and select **Run Native Application**, and click the **Remote Settings** tab.

   The **Remote Program** field should be set to **xterm -e %Application%** which causes the ouput to appear in an **xterm** on the Windows host.

5. Click **Run**.

In a moment, an **xterm** window should appear with the message **Hello World**, and close after ten seconds.

NOTE: If you get an error such as **connection to** *IP_address***:0.0 refused by server** it means your X server is not allowing the remote Linux host to connect.

Note that debug output will still appear in the Console view.

Building Locally on the Share

You can use the same share to build locally. For example, assuming you are configured with the same setup as the previous example, do the following:

1. Select **Project > Remote Connection**, select **Local Host** and click **Connect**.
2. Click the **Set Active Build Spec and Debug Mode** icon in the Project Explorer and set the active build spec to Windows GNU.
3. Right-click the **hello_world** project and select **Build Project**.The build is performed on the local Windows host.

Alternatively, you can configure additional connections to remote hosts and toggle between them by selecting them in **Project > Remote Connection.**

PART V

Target Management

13 Connecting to Targets 227

14 Connecting with USB 241

15 Connecting with TIPC 245

13 Connecting to Targets

13.1 Introduction 227

13.2 The Remote Systems View 228

13.3 Defining a New Connection 228

13.4 Establishing a Connection 230

13.5 Connection Settings 230

13.6 The Registry 237

13.1 Introduction

A Workbench *target connection* runs on a host and manages communications between host tools and the target system itself. A connection must be configured and established before the host tools can interact with the target.

All host-side connection configuration work and connection-related activity is done in the Remote Systems view. Connections are registered and made accessible to users by the Wind River Registry as described in *13.6 The Registry*, p.237. Connection data may be maintained in a central location and shared between hosts with the use of remote registries as described in*13.6.2 Remote Registries*, p.238.

13.2 The Remote Systems View

A connection to a Target Server must be defined and established before tools can communicate with a target system.

All host-side connection configuration work and connection-related activity is done in the Remote Systems view, by default at the bottom-left of Workbench. It is available in the Application Development and Device Debug perspectives. If the view is not visible, choose **Window > Show View > Remote Systems**, or, if not listed there, choose **Window > Show View > Other**.

The most import tasks in the Remote Systems view are:

- Defining new connections
- Connecting to targets
- Disconnecting from targets

Once you have connected to a target, more commands are enabled on the right-click context menu (see also *17. Launching Programs*).

13.3 Defining a New Connection

All connections types are defined from the Remote Systems view (see *13.2 The Remote Systems View*, p.228).

To open the **New Connection** wizard, either use the appropriate toolbar button, or right-click in the Target Manager and select **New > Connection**.

The first thing the **New Connection** wizard asks you to do is to select one of the following connection types:

- **Wind River Linux Application Core Dump Target Server Connection** – Create a new connection to a Linux 2.6 application core dump (see *19. Analyzing Core Files*).
- **Wind River Linux Host Development Target Server Connection** – Create a host development connection to a Linux 2.6 application.
- **Wind River Linux KGDB Connection** – Create a new kernel mode connection to a Linux 2.6 version kernel (see *4. Kernel Debugging (Kernel Mode)*).
- **Wind River RTCore Connection** – Create an RTCore Connection.

- **Wind River Linux User Mode Target Server Connection** – Create a usermode connection to the target.
- **Wind River QEMU Backend Connection** – Create a QEMU backend connection.
- **Wind River QEMU Connection** – Create a QEMU connection.
- **Wind River QEMU Linux User Mode Target Server Connection** – Create a QEMU target server connection (see *3. Creating and Debugging Platform Projects*).

Properties you set during the creation of a new connection using the **New Connection** wizard can be modified later by right-clicking the connection in the Remote Systems view and then selecting **Properties** from the context menu.

Note that if you change properties later, you will generally have to disconnect and reconnect in order for changes to take effect. To disconnect, right-click the connection and select **Disconnect**. To re-connect, right-click the connection and select **Connect**.

Depending on the type of connection you are making, Workbench presents different options and you enter different values. (For example, if you select **Wind River Linux Host Development Target Server Connection**, a **Browse** button appears for the usermode-agent field.) For details on all the options presented in the New Connection dialog, refer to *13.5 Connection Settings*, p.230.

Connecting to Core Dumps

Specify the name of the core dump file in **Core dump file**, and specify the path to the executable that caused the core dump in **Application image**.

For more information, refer to *19Analyzing Core Files*, p.309.

Connecting in Kernel Mode (KGDB Connection)

When configuring a kernel mode connection, you must first select the type of connection, which will be either by a serial line (RS232) or over Ethernet. An additional option is provided in case you are using a terminal server. A final option is for custom situations for which you should refer to Wind River Support.

Specify an IP address and the path to your kernel symbol file (**Kernel image**).

For more information on connecting in Kernel mode and how to configure the target to support KGDB, refer to *4. Kernel Debugging (Kernel Mode)*.

Connecting in User Mode

Specify the target's IP address and root file system when making a user mode connection. The target must be running the usermode agent.

Refer to *2. Introducing Application Development* for details on configuring the usermode agent and on making a user mode connection.

13.4 Establishing a Connection

Once you have created your application projects and defined connections, you will want to run, test, and debug the projects on your target. To do this, you first need to connect to the target.

Typically, after configuring a new connection in the New Connection dialog, the connection to the target is attempted automatically. If you do not want the connection to be attempted at that time, unselect **Immediately connect to target if possible** on the final dialog screen.

Connect to and disconnect from targets in the Remote Systems view by selecting a connection node and then using either the appropriate toolbar button, or by right-clicking and selecting **Connect**. See *13.2 The Remote Systems View*, p.228.

13.5 Connection Settings

When you have specified the type of connection to make, a new connection dialog box presents various parameters that you must configure.

Connection Template

For KGDB connections, select the type of connection that best describes how you are connecting to your target. Choose from the following connection types:

Linux KGDB via RS232
A serial cable connection.

Linux KGDB via Ethernet
An Ethernet connection.

Linux KGDB via Terminal Server
A connection in which you go through a terminal server to reach the target.

Custom Linux KGDB Connection
This template assumes no defaults.

Back End Settings

Back end
The **Back end** settings specify how a target server will communicate with a target.

Back end for a KGDB connection requires that you select from the following:

- **RS232** for a direct serial port connection (connection with a null-modem cable). You will then specify your serial line settings in the following dialog.
- **TCP** for connection to a terminal server with a direct serial connection.
- **UDP** for an Ethernet connection on the same subnet.
- **PIPE** is currently unsupported.

Back end for user mode connections may be set to **wdbrpc** or **wdbproxy**. **wdbrpc** supports any kind of IP connection (for example, Ethernet).

CAUTION: The target server *must* be configured with the same communication back end as the one built for the kernel image and used by the target agent.

CPU
Select **default from target** to have Workbench identify the target CPU for you, or select the target CPU from the drop-down menu.

Target Name/IP address
The **Name/IP Address** field specifies the network name or the IP address of the target hardware for networked targets.

Target File System and Kernel

Target File System and **Kernel** properties relate to the location of the target's root file system and the name and location of the target kernel. You specify the path to the kernel for KGDB connections.

Root File System
Enter the full path of the target's root file system as it is on the host. This file system is typically NFS-exported from the host.

Kernel Image
This is the full path name including the kernel symbol file, for example, **/home/wbuser/WindRiver/workspace/new_prj/export/vmlinux-symbols**.

Advanced Options (KGDB Only)

Backend Communication Log File
Enter or browse to the location of a file to store log information.

Target Plugin Pass-through Options
(See *4.6 Debugging User-Supplied Kernel Modules*, p.101.)

Advanced Target Server Options

These options are passed to the **tgtsvr** program on the command line (user mode only). Enter these options manually, or use the **Edit** button for GUI assisted editing.

Edit Target Server Options
Open the **Edit Target Server Options** window with the **Edit** button on the main wizard page. The options are subdivided into three tabbed groups: **Common**, **Memory**, and **Logging**.

The Common Tab

Timeout Options

Set how many seconds the target server will wait for an answer from the target before sending a timeout, how often to retry, and at what intervals it should ping the target, in seconds.

The Memory Tab

Memory Cache Size

To avoid excessive data-transfer transactions with the target, the target server maintains a cache on the host system. By default, this cache can grow up to a size of 1 MB.

A larger maximum cache size may be desirable if the memory pool used by host tools on the target is large, because transactions on memory outside the cache are far slower.

The Logging Tab

Options on the logging tab are used mainly for troubleshooting by Wind River support.

A maximum size can be specified for each of these files. If so, files are rewritten from the beginning when the maximum size is reached. If the file initially exists, it is deleted. This means that if the target server restarts (for example, due to a reboot), the log file will be reset.

For the WTX (Wind River Tool Exchange) log file, you can specify a *filter*, a regular expression that limits the type of information logged. In the absence of a filter, the log captures all WTX communication between host and target. Use this option in consultation with Customer Support.

Command Line

This will be filled-in based on the values you enter in this dialog box.

13.5.1 Target Operating System Settings

Clicking **Next** opens the Target Operating System Settings dialog box (Kernel mode only). Select the correct version of the Linux operating system, and specify

the path to the kernel symbol file (for example, **/home/wbuser/WindRiver/workspace/new_prj/export/vmlinux-symbols**).

13.5.2 Object Path Mappings

Clicking **Next** will open the Object Path Mappings dialog box (Kernel mode).

Workbench uses *Object Path Mappings* in two ways:

- They allow the debugger to find symbol files for processes created on the target by creating a correspondence between a path on the target and the appropriate path on the host.
- Workbench also uses object path mappings to calculate target paths for processes that you want to launch by browsing to them with a host file system browser.

By default, the debug server attempts to load all of a module's symbols each time a module is loaded. In the rare cases where you want to download a module or start a process without loading the symbol file, unselect **Load module symbols to debug server automatically if possible**.

13.5.3 Specifying an Object File

If you are loading object code on the target, or associating symbols with already loaded modules, you can specify the object file that you want the debugger to use.

1. Right-click a container in the Remote Systems view, then select **Load/Add Symbols to Debug Server**. A dialog appears with your connection and core already filled in.
2. To add a new object file to the **Symbol Files and Order** list, click **Add**. Navigate to the file, then click **Open**.
3. In the **Symbol Load Options** section, select **Specify base start address** or **Specify start address for each section**.
4. When you are finished, click **OK**.

For more information about the fields in this dialog, click in the Remote Systems view, then press the help key for your host.

Pathname Prefix Mappings

This maps target path prefixes to host paths. Always use full paths, not relative paths. For example, mapping **/** to **/opt/eldk/ppc_82xx/** tells the debugger that files accessible under **/** on the target can be found under **/opt/eldk/ppc_82xx/** on the host.

In most cases, Workbench provides correct defaults. If necessary, click **Add** to add new mappings, or select existing mappings and click **Edit** to modify existing mappings.

NOTE: You cannot edit the supplied default mappings.

Basename Mappings

Use square brackets to enclose each mapping of target file basenames (left element) to host file basenames (right element), separated by a semi-colon (;). Mapping pairs (in square brackets) are separated by commas. You can use an asterisk (*) as a wildcard.

For example, if debug versions of files are identified by the extension ***.unstripped**, the mapping **[*;*.unstripped]** will ensure that the debugger loads *yourApp***.unstripped** when *yourApp* is launched on the target.

13.5.4 Target State Refresh Page

Since retrieving status information from the target leads to considerable target traffic, this page allows you to configure how often and under what conditions the information displayed in the Remote Systems view is refreshed.

These settings can be changed later by right-clicking the target connection and selecting **Refresh Properties**.

Available CPU(s) on Target Board

Workbench can correctly identify the target CPU. In rare cases, a close variant might be misidentified, so you can manually set the CPU here.

Initial Target State Query and Settings

Specify whether Workbench should query the target on connect, on stopped events, and/or on running events. You can select all options if you like.

Target State Refresh Settings

Specify that the target state is refreshed only when you manually choose to do so, or if (and how often) the display should be refreshed automatically.

Listen to Execution Context Life-cycle Events

Specify whether Workbench should listen for life-cycle events or not.

13.5.5 Connection Summary Page (Target Server Connection)

This page proposes a unique but modifiable **Connection name**, and a **Summary** of name and path mappings. (Use the **Back** button to modify them.)

Shared
This option serves a dual purpose:

- When you define a target connection configuration, this connection is normally only visible for your user-id. If you define it as **Shared**, other users can also see the configuration in your registry, provided that they connect to your registry (by adding it as a remote registry on their computer, see *13.6.2 Remote Registries*, p.238).
- Normally, when you disconnect a target connection, the target server (and simulator) are killed because they are no longer needed. In a connection that is flagged as **Shared**, however, they are left running so that other users can connect to them. In other words, you can flag a connection as shared if you want to keep the target server (and simulator) running after you disconnect or exit Workbench.

Immediately connect to target if possible
If you do not want to connect to the target immediately, you can connect to the target later using one of the ways described in *18. Debugging Projects*.

If you have applications ready to run using the connection(s) you have just created, see *17. Launching Programs*.

13.6 The Registry

The Wind River Registry is a database of target servers, boards, ports, and other items used by Workbench to communicate with targets. For details about the registry, see the **wtxregd** and **wtxreg** entries in *Wind River Host Tools API Reference* in the online Help.

Before any target connections have been defined, the default registry—which runs on the local host—appears as a single node in the Remote Systems view. (Under Linux, the default registry is a target-server connection for Linux user mode.) Additional registries can be established on remote hosts.

Registries serve a number of purposes:

- The registry stores target connection configuration data. Once you have defined a connection, this information is persistently stored across sessions and is accessible from other computers.

  You can also share connection configuration data that is stored in the registry. This allows easy access to targets that have already been defined by other team members.

NOTE: Having connection configuration data does not yet mean that the target is actually connected.

- The registry keeps track of the currently running target servers and manages access to them.
- Workbench needs the registry to detect and launch target servers.

  If Workbench does not detect a running default registry at start-up, it launches one. After quitting, the registry is kept running in case it is needed by other tools.

13.6.1 Launching the Registry

To launch the default registry, open the **Target** menu or right-click in the Remote Systems view and select **Launch Default Registry**.

NOTE: These menu items are only available if the registry is not running, and the default registry host is identical to the local host.

The registry stores its internal data in the file *installDir*/**.wind/wtxregd.***hostname*. If this file is not writable on launch, the registry attempts to write to **/var/tmp/wtxregd.***hostname* instead. If this file is also not writable, the registry cannot start and an error message appears.

13.6.2 Remote Registries

If you have multiple target boards being used by multiple users, it makes sense to maintain connection data in a central place (the remote registry) that is accessible to everybody on the team. This saves everyone from having to remember communications parameters such as IP addresses and other settings for every board that they might need to use.

You might want to create a new *master registry* on a networked remote host that is accessible to everybody. To do so:

1. Workbench needs to be installed and the registry needs to be running on the remote host. The easiest way to launch the registry is to start and quit Workbench. However, you can also launch the **wtxregd** program from the command line.

   (For more information about **wtxregd**, see **Help > Help Contents > Wind River Documentation > References > Host API and Command References > Wind River Host Tools API Reference**.)

2. To use the remote registry on another host, right-click in the Remote Systems view, (see *13.2 The Remote Systems View*, p.228), then select **New > Registry** from the context menu.

3. In the dialog that appears, enter either the host name or the IP address of the remote host.

   Workbench immediately attempts to connect to the remote registry. If the host is invalid, or if no registry is identified on the remote host, this information is displayed in the Remote Systems view. A valid connection will display the registry in the Remote Systems view and any active connections will be shown. Connect to the target just as you would to a target in your local registry.

13.6.3 Shutting Down the Registry

Because other tools use the registry, it is not automatically shut down when you quit Workbench. Before updating or uninstalling Workbench (or other products that use the registry), it is advisable to shut down the registry so that the new one starts with a fresh database. To shut down the registry:

- On Windows, right-click the registry button in the system tray, and choose **Shutdown**.
- On Linux and UNIX, execute **wtxregd stop**, or manually kill the **wtxregd** process.

If you want to migrate your existing registry database and all of your existing connection configurations to the new version, make a backup of the registry data file *installDir*/**.wind/wtxregd.***hostname* and copy it to the corresponding new product installation location.

13.6.4 Changing the Default Registry

Normally, the default registry runs on the local computer. You can change this if you want to force a default remote registry (see *13.6.2 Remote Registries*, p.238). To do this on Linux and UNIX, modify the **WIND_REGISTRY** environment variable. To do this on Windows, under the Windows Registry **HKEY_LOCAL_MACHINE** node, modify the field **Software\Wind River Systems\Wtx***N.N***\WIND_REGISTRY**.

14

Connecting with USB

14.1 Introduction 241

14.2 Configuring a Target for USB Connection 242

14.3 Configuring a Host for USB Connection 243

14.1 Introduction

You can make a USB connection between a Workbench host and a target that supports the USB driver. Once you have established the connection, you can use it to debug applications just as you would with an Ethernet connection.

14.2 Configuring a Target for USB Connection

The way you configure your target depends on the host platform.

Target Configuration for a Linux Kernel 2.6 Host

The target kernel must support the USB device driver (USB Ethernet gadget). It may be installed as a module or it may be built into the kernel (this is the default for Wind River Linux kernels).

Determining Driver Status

To see if the USB Ethernet gadget is installed as a kernel module, examine the **lsmod** command output. To see if the gadget is built into your kernel, search the **dmesg** output for something similar to the following:

```
usb0: Ethernet Gadget, version: Equinox 2004
usb0: using pxa27x_udc, OUT ep3out-bulk IN ep2in-bulk STATUS ep1in-int
usb0: MAC 9a:8c:21:b0:9f:42
usb0: HOST MAC 26:06:24:43:81:a0
```

(Your MAC and HOST MAC address will be different.)

If **lsmod** shows the module installed or **dmesg** shows the gadget driver built in, you can proceed to *Configuring the USB Interface*, p.242.

Adding Driver Support

To add USB gadget driver support, build it into the kernel or add it as a module with **Device Drivers > USB support > USB Gadget Support > Support for USB Gadgets** using **make linux.xconfig** or the Wind River platform's **Kernel Configuration** node.

To install the kernel module, enter:

```
# modprobe g_ether
```

Configuring the USB Interface

With the kernel correctly configured and running, configure the **usb0** interface, for example:

```
# ifconfig usb0 10.0.0.150
```

where **10.0.0.150** is the IP address assigned to the target.

Target Configuration for a Windows Host

Reconfigure your kernel to enable RNDIS support as follows.

```
$ cd dist
$ make linux.xconfig
```

Go to **Device Drivers > USB Support > USB Gadget Support** and select **RNDIS Support**. Save and exit.

```
$ make linux.build
```

With the kernel correctly configured and running, configure the **usb0** interface, for example:

```
# ifconfig usb0 10.0.0.150
```

where **10.0.0.150** is the IP address assigned to the target.

14.3 Configuring a Host for USB Connection

The configuration procedure depends on the host OS.

Linux 2.6 Host Configuration

Driver Support

The host should have the **g_ether** module installed. Refer to *Target Configuration for a Linux Kernel 2.6 Host*, p.242 for details on installing the **g_ether** module.

Interface Configuration

For a Linux 2.6 host, configure and test the USB interface as follows.

1. Connect a USB cable with the host connector attached to the Workbench host and the peripheral connector attached to the target.
2. Configure the USB interface on the host as follows.

   ```
   # ifconfig usb0 10.0.0.151
   ```

   In this example, **10.0.0.151** is the address assigned to the host on the USB connection.

3. Verify that the connection is working with a **ping** command to the target:

```
$ ping 10.0.0.150
```

A successful ping of the target over the USB connection shows that your host and target are properly connected.

Windows Host Configuration

You must install the **linux.inf** file on the windows host. The file is available in the **Documentation** directory of most kernels including all Wind River Linux kernels.

To install the file, first convert it to the DOS text file format with the **unix2dos** command:

```
$ unix2dos linux.inf
```

Connect a USB cable with the host connector attached to the Workbench host and the peripheral connector attached to the target.

When you attach the USB cable to the Windows host, Windows will recognize new hardware and start the new hardware wizard. Follow the prompts and install the **linux.inf** file.

15 Connecting with TIPC

15.1 Introduction 245
15.2 Configuring TIPC Targets 246
15.3 Configuring a TIPC Proxy 248
15.4 Configuring Your Workbench Host 250
15.5 Using usermode-agent with TIPC 251

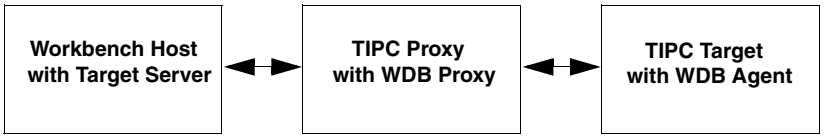
15.1 Introduction

This chapter describes how to configure Linux TIPC targets and your Workbench host to support debugging. For detailed information about TIPC, see the official TIPC project Web site at **http://tipc.sourceforge.net/**.

The transparent inter-process communication (TIPC) infrastructure is designed for inter-node (cluster) communication. Targets located in a TIPC cluster may not have access to standard communication links or may not be able to communicate with hosts not located on the TIPC network. Because of this, host tools used for development may not be able to access those targets and debug them without special tools. To solve this communication problem between the TIPC target and TCP/IP hosts, Wind River provides the **wrproxy** process, which acts as a gateway between the host and the target. (For a more generalized use of **wrproxy**, see *D. Configuring a Wind River Proxy Host*.)

A basic diagram of a Workbench host configured to debug a TIPC target is shown in Figure 15-1.

Figure 15-1 **Workbench Host, Proxy, and TIPC Target**



The Workbench host communicates using UDP, the TIPC target communicates using TIPC, and the proxy translates between them.

Note that the functions of the three network hosts shown in Figure 15-1 may be combined in different ways, for example, the WDB proxy and WDB agent may both reside on a single target. You may even configure your Workbench host to support all functions if you want to test your debug capabilities in native mode before configuring external TIPC targets.

The following sections describe how to configure TIPC targets, configure a proxy, and configure your Workbench host to support debugging over TIPC.

15.2 Configuring TIPC Targets

To configure TIPC targets, you must install the TIPC kernel module on them. To configure them to communicate with Workbench, you must also run the WDB agent on them.

Note that TIPC communication between nodes in a cluster does not require UDP or TCP/IP networking services so those functions do not need to be included with the kernel, enabling a smaller kernel with fast, intra-node (TIPC) communication. For the TIPC-configured node to communicate with the Workbench host, however, a proxy must be provided that is capable of both TIPC and UDP communication capabilities. The proxy may be provided by one of the cluster nodes, a separate host, or the Workbench host itself as described in *15.3 Configuring a TIPC Proxy*, p.248.

15.2.1 Installing the TIPC Kernel Module

If you are using a Wind River Linux platform, the **tipc.ko** kernel module is supplied. If you do not have the kernel module, you can download the source from **http://sourceforge.net/projects/tipc** and then build it based on the instructions in the downloaded **README** file.

Once you have the kernel module, install it and configure it on the target as follows.

1. Load the TIPC kernel module:

   ```
   # insmod /lib/modules/2.6.SubVersion-gpp/net/tipc.ko
   ```

   (The example shown is for the module supplied with one version of the Wind River Linux platform. The location of your **tipc.ko** kernel module may differ.)

2. Set the local TIPC address:

   ```
   # tipc-config -a=1.1.1 -be=eth:eth0
   ```

   (Your actual command will differ if your network device is not **eth0** or if you chose an address different from **1.1.1**.)

3. Check that everything is configured properly:

   ```
   # tipc-config -a
   ```

   The output should display your current TIPC address, for example, **1.1.1**.

15.2.2 Running the usermode-agent

You must run **usermode-agent** on each target you want to reach. Find the correct agent for your target architecture in *installDir*/**linux-2.x/usermode-agent/1.1/bin/***arch*. For example, the correct agent for the PPC architecture, when your *installDir* is **WindRiver**, is **WindRiver/linux-2.x/usermode-agent/1.1/bin/ppc/usermode-agent**.

To launch **usermode-agent** on the target:

1. Copy it to the target.
2. Change directory (**cd)** to the directory where it is located.
3. Enter the following command:

   ```
   $ ./usermode-agent -comm tipc &
   ```

(For help in setting up usermode-agent to run when the target boots, see *G. Starting usermode-agent at Boot Time*.)

15.3 Configuring a TIPC Proxy

The WDB proxy enables communication between the Workbench host and the TIPC target. The target server on the Workbench host (see *15.4 Configuring Your Workbench Host*, p.250) instructs the proxy agent to communicate using TIPC with a specified TIPC target address.

The WDB proxy agent is the **wrproxy** command (or **wrproxy.exe** with Windows). The host that runs **wrproxy** must have TIPC capability. To configure TIPC capability, install the TIPC kernel module (see *15.2.1 Installing the TIPC Kernel Module*, p.247), or build TIPC into the kernel and reboot it. When you have TIPC support in the kernel, configure the host with a TIPC address that is different from target TIPC addresses using **tipc-config** (see *15.2.1 Installing the TIPC Kernel Module*, p.247).

The **wrproxy** command is located in *installDir***/workbench-***version***/foundation/***version***/x86-linux2/bin/**. Enter the following command on the TIPC-capable network host that is to serve as the proxy between Workbench and the TIPC target:

```
$ wrproxy &
```

You can also use the **-p** *port* option to specify a different TCP port number for **wrproxy** to listen to (default **0x4444**), the **-V** option for verbose mode, or the **-h** option to get command help.

NOTE: If you specify a port other than the default port for the proxy, then you must specify the same port when configuring the target server as described in *15.4 Configuring Your Workbench Host*, p.250.

Figure 15-2 illustrates a configuration in which the proxy agent runs on the same host as Workbench. Figure 15-3 illustrates a configuration in which the proxy agent runs on one of the nodes in a cluster. Another example might be a separate host that runs **wrproxy**, between the targets in the cluster and the Workbench host.

Figure 15-2 **TIPC Configuration with WDB Proxy Agent on Workbench Host**

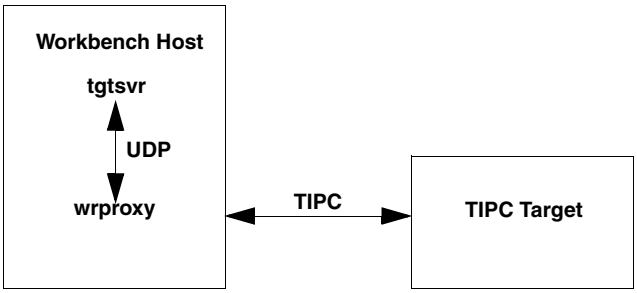
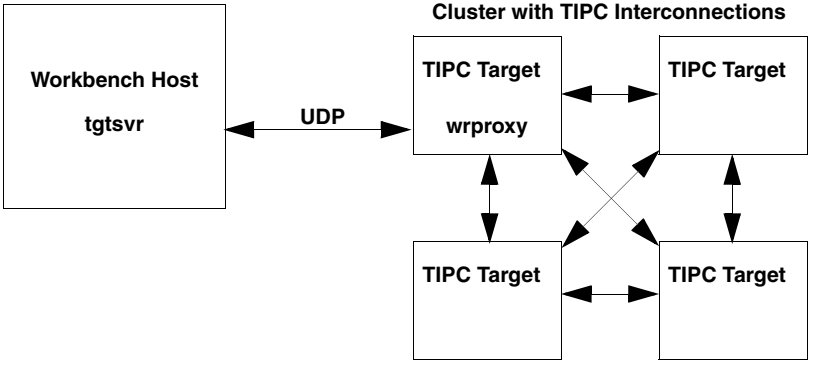


Figure 15-3 **TIPC Configuration with WDB Proxy Agent on Cluster Target**



15.4 Configuring Your Workbench Host, p.250 describes how to configure the target server on the Workbench host to connect to the proxy agent and reach the TIPC target that you want to connect to.

15.4 Configuring Your Workbench Host

Use the **tgtsvr** command to connect to the proxy for communication with a TIPC target. The following command shows the TIPC options to use:

```
tgtsvr [-V] -B wdbproxy -tipc -tgt targetTipcAddress ProxyIpAddres
```

For example, to connect to a target with a TIPC address of 1.1.8 using a proxy with the IP address 192.168.1.5, use the following command:

```
$ tgtsvr -B wdbproxy -tipc -tgt 1.1.8 192.168.1.5
```

Additional Information

A fuller syntax for the **tgtsvr** command is:

```
tgtsvr [-V] -B wdbproxy -tipc -tgt targetTipcAddress [-tipcpt tipcPortType -tipcpi
tipcPortInstance] wdbProxyIpAddress|name
```

Table 15-1 explains the italicized parameter values in the command.

Table 15-1 **TIPC-Specific Parameter Values for Starting a Target Server**

Parameter	Value
targetTipcAddress	The TIPC address of the target with the **TIPC network stack**. For example: **1.1.8**.
tipcPortType	The TIPC port type to use in connecting to the WDB target agent. The default port type for the connection is **70**. You should accept the default port unless it is already in use.
tipcPortInstance	The TIPC port instance to use in connecting to the WDB target agent. The default port instance for the connection is **71**. You should accept the default port instance unless it is already in use.
wdbProxyIpAddress|*name*	The IP address or DNS name of the target with WDB Agent Proxy.

Note that if you change the default TIPC port configuration, you must also change the default TIPC port for the **usermode-agent** as described in *15.5 Using usermode-agent with TIPC*, p.251.

Alternatively, you can use the Workbench GUI to configure the host. Select **wdbproxy** as the backend when you create a new connection in the Remote Systems view and then fill in the fields with the values you would supply as command line arguments. The command line that is created at the bottom of the GUI should be similar to the example shown in this section.

15.5 Using usermode-agent with TIPC

This section explains the several possible options available when launching the usermode agent.

The listening port is the port used by the usermode agent to communicate with the target server on the host machine. If you change the listening port on the usermode agent side, then you have to specify the same port number to the target server.

Port option

The port option is:

-p or **-port 0x***pppp* (UDP) | *xxxx*:**yyyy** (TIPC)

This option allows you to select an alternate listening port for the usermode agent.

Two network connection types are supported:

- UDP – this is the default connection type. If you do not specify a particular type of network connection, UDP is used.

  If you do not want to use the default UDP port (0x4321), you can choose and set the one you want using this option. The port number can be entered in either decimal or hexadecimal format. To set the port number using the hexadecimal format, use the **0x**%*x* format where %*x* represents the port number in hexadecimal base.

  For example, to launch the usermode agent using UDP and port 6677:

  ```
  $ usermode-agent -p 6677
  ```

  or

  ```
  $ usermode-agent -p 0x1A15
  ```

- TIPC – this is the TIPC network connection. If you do not want to use the default TIPC port type (70) and TIPC port instance (71) then you can choose

and set the ones you want using this option. The port numbers can be entered in either decimal or hexadecimal format. To set the port numbers using the hexadecimal format, you need to use the **0x**%*x* format where %*x* represents the port number in hexadecimal base.

To launch the usermode agent using TIPC and port type 1234, port instance 55:

```
$ usermode-agent -p 1234:55
```

or

```
$ usermode-agent -p 0x4D2:37
```

Communication Option

The communication option allows to specify which kind of connection will be used for connection between target server and usermode agent.

Comm option is:

```
-comm serial | tipc
```

If the serial option is set then you can also specify the serial link device to use rather than the default one (/**dev/ttyAMA1**) and the baud speed for the serial link (115200 is the default baud speed).

To set a different device for the serial link connection, the flag **-dev** has to be used with the **-comm** serial option. For the baud speed, you need to set the **-baud** option combined with the **-comm** serial option.

Example

To launch the usermode agent using serial link connection and serial device **/dev/ttyS0**:

```
$ usermode-agent -comm serial -dev /dev/ttyS0
```

Example

To launch the usermode agent using serial link connection with default serial device and baud speed of 19200:

```
$ usermode-agent -comm serial -baud 19200
```

Example

To launch the usermode agent using serial link connection with serial device **/dev/ttyS0** and baud speed of 19200:

```
$ usermode-agent -comm serial -dev /dev/ttyS0 -baud 19200
```

If the **tipc** option is set then you can also specify the port type (default is 70) and port instance (default is 71) of the TIPC connection.

To set a different port type or instance, use the flag **-tipcpt** or **-tipcpi**, in either decimal or hexadecimal format.

Examples

To launch the usermode agent using TIPC network connection with default port type and default port instance:

```
$ usermode-agent -comm tipc
```

To launch the usermode agent using TIPC network connection with specific port type 123 and specific port instance 456:

```
$ usermode-agent -comm tipc -tipcpt 123 -tipcpi 456
```

Daemon mode

The **-daemon** option lets the usermode agent become a daemon after all initialization functions are completed. The output message, if any, are still reported on the device where the process has been started.

Environment Inheritance

The **-inherit-env** option makes all the child processes inheriting the environment from the parent environment. Since the usermode agent is the father of all the processes, then the processes will inherit the shell environment from which the usermode agent has been launched.

No Thread Support (Linux Thread Model Only)

The **-no-threads** option allows you to use the usermode agent on a kernel using Linux threading model even if the **libpthread** library is stripped. Basically, the **libpthread** library is used by the usermode agent to detect thread creation, destruction and so on. On a kernel using Linux threading model, if the **libpthread** library is stripped then the multithread debug would not be reliable so, by default, the usermode agent exit if this option is not set, to ensure a reliable debug scenario.

This option is useless if your kernel is running using NPTL threading model.

Other Options

The **-v** option displays version information about the usermode agent, that is, build and release information.

The **-V** option set the usermode agent to run in verbose mode. This is useful to have the listening information: port number, listening connection type and the target server connection to this usermode agent.

The **-help** or **-h** option displays all the possible startup options for the usermode agent.

PART VI

Debugging

16 Managing Breakpoints 257

17 Launching Programs 265

18 Debugging Projects 287

19 Analyzing Core Files 309

16 Managing Breakpoints

16.1 Introduction 257

16.2 Types of Breakpoints 258

16.3 Manipulating Breakpoints 262

16.1 Introduction

Breakpoints allow you to stop a running program at particular places in the code or when specific conditions exist.

This chapter shows how to use the Breakpoints view to keep track of all breakpoints, along with their conditions (if any).

You can create breakpoints by double-clicking or right-clicking in the Editor's left vertical ruler (also known as the *gutter*), or by opening the various breakpoint dialog boxes from the pull-down menu in the Breakpoints view.

If you are using the CDT/GDB debugger for native-mode (self-hosted) applications (as described briefly in *2.2.3 Debugging Self-Hosted Applications with the CDT/GDB Debugger*, p.35), note that some breakpoint actions always create breakpoints exclusively for the Wind River debugger. This applies to all actions in the Breakpoints and Tracepoints submenus. If a debugger has at least one debug session active, the active debugger is chosen. If there are mixed debug sessions, the

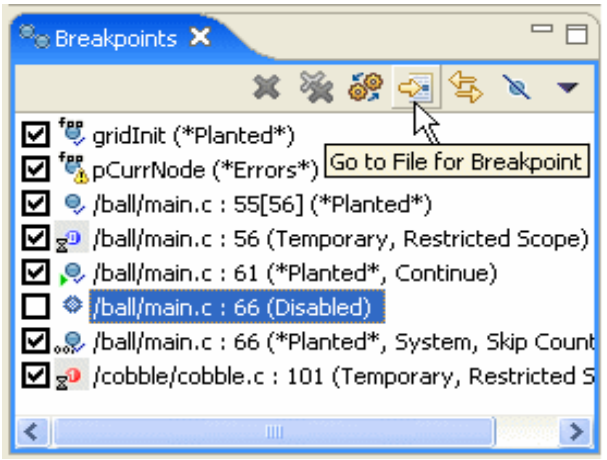
breakpoints are seen by the Wind River debugger. In particular, the Wind River debugger is assumed or chosen:

- If no project is associated with the open file, because CDT does not support project-less debugging.
- If there is no active debugger, and the target seems to be a device. Otherwise if the project is a CDT project, a CDT breakpoint will be created.
- If a Wind River perspective is open (such as Application Development, Device Debug, On Chip Debug, or Hardware Debug), otherwise the CDT debugger is chosen.

16.2 Types of Breakpoints

Figure 16-1 shows the Breakpoints view with various types of breakpoints set.

Figure 16-1 **Breakpoints View**


Breakpoints
gridInit (*Planted*)
pCurrNode (*Errors*)
Go to File for Breakpoint
/ball/main.c : 55[56] (*Planted*)
/ball/main.c : 56 (Temporary, Restricted Scope)
/ball/main.c : 61 (*Planted*, Continue)
/ball/main.c : 66 (Disabled)
/ball/main.c : 66 (*Planted*, System, Skip Count
/cobble/cobble.c : 101 (Temporary, Restricted S

See the sections below for when and how to use each type of breakpoint.

16.2.1 Line Breakpoints

Set a line breakpoint to stop your program at a particular line of source code.

To set a line breakpoint with an unrestricted scope (that will be hit by any process or task running on your target), double-click in the left gutter next to the line on which you want to set the breakpoint. A solid dot appears in the gutter, and the Breakpoints view displays the file and the line number of the breakpoint. You can also right-click in the gutter and select **Add Global Line Breakpoint**.

To set a line breakpoint that is restricted to just one task or process, right-click in the Editor gutter and select **Add Breakpoint for** *"selected thread"*. If the selected thread has a color in the Debug view, a dot with the same color will appear in the Editor gutter with the number of the thread inscribed inside it.

16.2.2 Expression Breakpoints

Set an expression breakpoint using any C expression that will evaluate to a memory address. This could be a function name, a function name plus a constant, a global variable, a line of assembly code, or just a memory address. Expression breakpoints appear in the Editor's gutter only when you are connected to a task.

Breakpoint conditions are evaluated after a breakpoint is triggered, in the context of the stopped process. Functions in the condition string are evaluated as addresses and are not executed. Other restrictions are similar to the C/C++ restrictions for calculating the address of a breakpoint using the Expression Breakpoint dialog box.

16.2.3 Hardware Breakpoints

Some processors provide specialized registers, called debug registers, which can be used to specify an area of memory to be monitored. For instance, IA-32 processors have four debug address registers, which can be used to set data breakpoints or control breakpoints.

Hardware breakpoints are particularly useful if you want to stop a process when a specific variable is written or read. For example, with hardware data breakpoints, a hardware trap is generated when a write or read occurs in a monitored area of memory. Hardware breakpoints are fast, but their availability is machine-dependent. On most CPUs that do support them, only four debug registers are provided, so only a maximum of four memory locations can be watched in this way.

There are two types of hardware breakpoints:

- A hardware *data breakpoint* occurs when a specific variable is read or written.
- A hardware *instruction breakpoint* or *code breakpoint* occurs when a specific instruction is read for execution.

Once a hardware breakpoint is trapped—either an instruction breakpoint or a data breakpoint—the debugger will behave in the same way as for a standard breakpoint and stop for user interaction.

Adding Hardware Instruction Breakpoints

There two ways to add a new hardware instruction breakpoint:

- In the gutter (grey column) on the left of the source file, right-click and select **Add Hardware Code Breakpoint**.
- Or, double-click in the gutter to add a standard breakpoint and then, in the breakpoint view, right-click the breakpoint you've just added and select **Properties**. In the last pane (**Hardware**) of the **Properties** dialog box select **Enable Hardware Breakpoint**.

Adding Hardware Data Breakpoints

Set a hardware data breakpoint when:

- The debugger should break when an event (such as a read or write of a specific memory address) or a situation (such as data at one address matching data at another address) occurs.
- Threads are interfering with each other, or memory is being accessed improperly, or whenever the sequence or timing of runtime events is critical (hardware breakpoints are faster than software breakpoints).

To add a hardware data breakpoint, go to the breakpoint view, then click the down arrow in the top right of this view, and select **Add Data Breakpoint** to display the hardware data breakpoint dialog box. You are presented with four tabs. Use the **General** tab to enter the variable you want to monitor in the **Address Expression** box. Use **Status** and **Scope** tabs the same way for hardware code breakpoints. Use the **Hardware** tab to bold fields that you can check to make a selection from the drop-down list, for example to choose the access size (**Byte**, **Half-Word**, or **Word**) and the access type you want to monitor for this variable.

Disabling and Removing Hardware Breakpoints

You can disable and remove hardware breakpoints in the same ways that you disable and remove standard breakpoints.

Converting Breakpoints to Hardware Breakpoints

To cause the debugger to request that a line or expression breakpoint be a hardware code breakpoint, select the **Hardware** check box on the **General** tab of the **Line Breakpoint** or **Expression Breakpoint** dialog boxes.

This request does not guarantee that the hardware code breakpoint will be planted; that depends on whether the target supports hardware breakpoints, and if so, whether or not the total number supported by the target have already been planted. If the target does not support hardware code breakpoints, an error message will appear when the debugger tries to plant the breakpoint.

NOTE: Workbench will set only the number of code breakpoints, with the specific capabilities, supported by your hardware.

NOTE: If you create a breakpoint on a line that does not have any corresponding code, the debugger will plant the breakpoint on the next line that does have code. The breakpoint will appear on the new line in the Editor gutter.

In the Breakpoints view, the original line number will appear, with the new line number in square brackets **[ ]** after it. See the third breakpoint in Figure 16-1.

Comparing Software and Hardware Breakpoints

Software breakpoints work by replacing the destination instruction with a software interrupt. Therefore it is impossible to debug code in ROM using software breakpoints.

Hardware breakpoints work by comparing the break condition against the execution stream. Therefore they work in RAM, ROM or flash.

Complex breakpoints involve conditions. An example might be, “Break if the program writes *value* to *variable* if and only if *function_name* was called first.”

16.3 Manipulating Breakpoints

Now that you have an understanding of the different types of breakpoints, this section will show you how to work with them.

16.3.1 Exporting Breakpoints

To export breakpoint properties to a file:

1. Select **File > Export > Export Breakpoints**, then click **Next**. The Export Breakpoints dialog box appears.
1. Select the breakpoint whose properties you want to export, and type in a file name for the exported file. Click **Finish**.

16.3.2 Importing Breakpoints

To import breakpoint properties from a file:

1. Select **File > Import > Import Breakpoints**, then click **Next**. The Import Breakpoints dialog box appears.
2. Select the breakpoint file you want to import, then click **Next**. The Select Breakpoints dialog box appears.
3. Select one or more breakpoints to import, then click **Finish**. The breakpoint information will appear in the Breakpoints view].

16.3.3 Refreshing Breakpoints

Right-clicking a breakpoint in the Breakpoints view and selecting **Refresh Breakpoint** causes the breakpoint to be removed and reinserted on the target. This is useful if something has changed on the target (for example, a new module was downloaded) and the breakpoint is not automatically updated.

To refresh all breakpoints in this way, select **Refresh All Breakpoints** from the Breakpoints view toolbar drop-down menu.

16.3.4 Disabling Breakpoints

To disable a breakpoint, clear its check box in the Breakpoints view. This retains all breakpoint properties, but ensures that it will not stop the running process. To re-enable the breakpoint, select the box again.

16.3.5 Removing Breakpoints

There are several ways to remove a breakpoint:

- Right-click it in the Editor gutter, select **Remove Breakpoint**.
- Select it in the Breakpoints view, click the **Remove** button.
- Right-click it in the Breakpoints view, select **Remove**.

17 Launching Programs

17.1 Introduction 265

17.2 Creating a Launch Configuration 266

17.3 Remote Java Launches 272

17.4 Launching Programs Manually 275

17.5 Controlling Multiple Launches 276

17.6 Launches and the Console View 281

17.7 Attaching the Debugger to a Running Process 283

17.8 Attaching to the Kernel 284

17.9 Suggested Workflow 285

17.1 Introduction

Whenever you run a process or task from Workbench, a *launch configuration* is automatically created for you. A launch configuration is like a named script that captures the whole process of building, connecting a target, running, and possibly attaching a debugger. You can rerun your previous launches at any time by clicking a single button.

This chapter explains how to edit and fine-tune your launch configurations to provide a tight edit-compile-debug cycle, as well as how to manually attach the debugger to tasks and processes. See *3.8.1 Accepting the Automatic Debug Launch Configuration*, p.69 for automatic launch configuration support.

17.2 Creating a Launch Configuration

A launch configuration is similar to a macro because it allows you to group together the actions required to start a program. Then, with a single click, you can connect to your target, start your process, and if you wish, attach the debugger. Your configurations are stored persistently, and can be shared with your team.

Launch configurations can be run in one of two modes:

- *Run-mode* connects to your target, then launches a process.
- *Debug-mode* is like run-mode, but it automatically attaches the Wind River debugger after completing all other actions.

The same launch configuration can be executed in either mode.

To create a launch configuration:

1. Select **Run > Run** or **Run > Debug**. The **Create, manage, and run configurations** dialog box appears.
1. From the **Configurations** list, select the type of configuration you want to create, **Attach to Target** or **Process on Target** (explained below). Click **New**.
2. Once you click **New**, tabs appear and display the appropriate fields and options for your configuration type.

17.2.1 Editing an Attach to Target Launch Configuration

You do not create **Attach to Target** launch configurations manually. Instead, these configurations are created automatically when you attach to a process or kernel task from the Remote Systems view.

Attach to Target launch configurations are special in some ways:

- They do not actually run something but just connect a target and attach the debugger to some context that must already exist.
- They are visible only in Debug mode.
- **Attach to Target** launches cannot be created manually. They are created automatically when you attach the debugger to a context using the Remote Systems view.

Once an **Attach to Target** launch is created, you can review and edit it in the Launch Configurations dialog box.

The Main Tab

The properties in the Main tab are for review only and cannot be changed.

What you can do is:

- Review your existing attaches and delete those that you no longer need.
- Rename your attaches and, if you think they are valuable, put them into your **Favorites** menu using the **Common** tab.
- Change the mapping between source paths compiled into your objects and source paths in your workspace by editing the Source Locator information in the **Sources** tab.
- Change the **Projects to Build** settings for the launch.

The Projects to Build Tab

The **Projects to Build** tab displays the projects that Workbench will build before launching the process in this configuration. To disable this, clear **Window > Preferences > Run/Debug > Launching > Build (if required) before launching**.

To add to the list, click **Add Project**, select one or more projects from the dialog box, then click **OK**.

To rearrange the build order, select a project then click **Up**, **Down**, or **Remove**.

Note that the Projects to Build list takes project-subproject relationships from the Project Explorer into account. Thus, when **myLib** is a subproject of **myProj** and you choose to add **myProj** to the list, you cannot add **myLib** to the list as well because it will be built automatically when you build **myProj**. Adding **myLib** as well would be redundant and is thus disabled.

The Source Tab

The **Sources** tab displays the order in which locations will be searched for source files during debugging. The search order is determined by a location's position in the list.

Configuring the Source Lookup Path is optional, and is only necessary if the build-target was compiled on a different host. See *18.2.9 Changing Source Lookup Settings*, p.297 for more information about the source locator.

1. On the **Sources** tab, click **Add** to configure the source lookup path.
2. Select the type of source to add to the lookup path.
3. Once you add a new source to the lookup path, you can adjust its position in the search order by clicking **Up** or **Down** to change its position in the list.

The Common Tab

The **Common** tab allows you to specify who can access this configuration, and whether it appears in the Workbench toolbar menu.

1. If this launch configuration is shared with others on your team, click **Shared**, then type or browse to the directory where the shared configuration is located.
2. If you want to be able to access this launch configuration from the **Debug** favorites menu (the drop-down menu next to the bug button on the Workbench toolbar), select **Debug** in the **Display in favorites menu** box.
3. If you want the process to launch in the background, ensure that box is selected.
4. Click **Apply** to save your settings but leave the dialog box open, click **Close** to save your launch configuration for later use, or click **Debug** to launch it now.

17.2.2 Creating a Process Launch Configuration

Once you click **New**, the **Main**, **Projects to Build**, **Debug Options** (in debug-mode), **Source**, and **Common** tabs appear.

The **Name** of the build target you selected in the **Project Explorer** appears at the top of the dialog box in the form *name - connection_name*. If you did not select a build target, or want to modify the name that appears, type a descriptive **Name** for your new launch configuration.

17.2.3 The Main Tab

The **Main** tab displays information about the output file that you want to download and run during the launch.

1. On the **Main** tab of the dialog box, keep the default **Connection registry** and **Connection to use** settings, or if you have more than one registry or connection defined in the Remote Systems view, you may select a different one from the pull-down list.

   To create a new registry or connection type, click **Add**.

2. The **Entry Point**, **Arguments**, **Priority**, and **Stack size** fields are only active when you are connected to a target. To retrieve the connection-specific properties from the target, and adjust them if necessary, click **Connect**. Once your target is connected, you can also click **Edit** to open the **Advanced Options** dialog box.

3. To adjust the options field, click **Select**. The **Options** dialog box appears.

17.2.4 The Projects to Build Tab

You can specify that other projects should be built (if necessary) before launching the current project with the **Projects to Build** tab. This only applies if you have checked **Build (if required) before launching** in the **Window > Preferences > Run/Debug > Launching** dialog box.

Click **Add project** to select one or more existing projects. You can change the order in which they are built with the **Up** and **Down** buttons, or click **Remove** to remove them from the list. Then click **Apply** and **Close**. When this configuration is launched, those projects will first be built first (if required), in the specified order, prior to launch of this project.

17.2.5 The Debug Options Tab

The **Debug Options** tab only appears for launch configurations in debug-mode. With **Break on Entry** checked and **main** entered in the box, the process will break at launch on the entry to the **main( )** routine for debugging operations.

17.2.6 The Source Tab

The **Source** tab displays the search order for source files during debugging. The search order is determined by a location's position in the list.

Configuring the **Source Lookup Path** is optional, and is only necessary if the build-target was compiled on a different host. See *18.2.9 Changing Source Lookup Settings*, p.297 for more information about the source locator.

1. On the **Source** tab, click **Add** to configure the source lookup path.
2. Select the type of source to add to the lookup path; see *18.2.9 Changing Source Lookup Settings*, p.297 for a description of each type.
3. Once you add a new source to the lookup path, you can adjust its position in the search order by clicking **Up** or **Down** to change its position in the list.

17.2.7 The Common Tab

The **Common** tab allows you to specify whether this launch configuration is **Local** or **Shared** (local is the default), whether you want to access it from the Workbench toolbar buttons, and if the program should be launched in the background.

1. If this launch configuration is shared with others on your team, click **Shared**, then type in or browse to the directory where the shared configuration is located.
2. If you want to be able to access this launch configuration from the **Debug** favorites menu (the drop-down list next to the bug button on the Workbench toolbar), select **Debug** in the **Display in favorites menu** box.
3. If you want the process to launch in the background, ensure that box is checked.
4. Click **Apply** to save your settings, but leave the dialog box open, click **Close** to save your launch configuration for later use, or click **Debug** to launch it now.

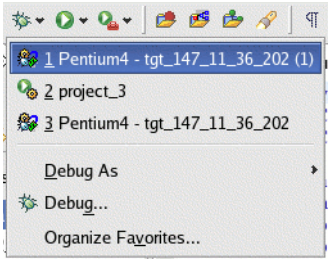
17.2.8 Using Launch Configurations to Run Programs

In a typical development scenario, you will run the same application many times in a single debugging session. After creating a launch configuration, you can click the **Debug** button to run a process and attach the debugger in a few seconds.

To launch a recently run process:

1. Click the pull-down arrow next to the **Debug** button and select the process from the configuration list. Figure 17-1.

Figure 17-1 **Debug Launch List**



If you ran the configuration recently, it will appear on the menu. If you selected **Debug** from the **Display in favorites menu** list (see *The Common Tab*, p.270) for a configuration, it will always appear on the list, whether you have run it recently or not.

2. To run a configuration not listed on the menu, click **Debug**, choose the configuration from the **Configurations** list, and click **Debug**.

NOTE: Different launches are associated with perspectives. To change the perspective go to **Run > Debug**. Select a launch type on the left and then you can select the desired perspective for either **Run > Run** or **Run > Debug**.

Increasing the Launch History

Workbench stores a history of previously launched configurations. The default length of the launch history is 10, but you can increase the history length by selecting **Window > Preferences > Run/Debug > Launching** and increasing the number in the **Size of recently launched applications list** field.

Troubleshooting Launch Configurations

If you press the **Debug** button (or click the **Debug** button from the **Launch Configuration** dialog box) and get a "Cannot create context" error, check the **Exec Path** on the **Main** tab of the **Debug** dialog box (see page 272) to be sure it is correct. Also check your Object Path Mappings.

17.3 Remote Java Launches

There are four Java launch configuration types in Workbench:

- **Java Applet**
- **Java Application**
- **Remote Java Application**
- **Remote Java Launch and Connect**

The first three are standard Eclipse-with-JDT launch configuration types. **Java Applet** and **Java Application** are for native mode debugging of Java applets and Java applications. The **Remote Java Application** launch configuration type is for connecting to an already running Java application on a remote target. It does not launch the application on the target. You must start the application through whatever means are available and apply the necessary options for remote debugging. Also, the I/O capabilities of the Console view are not available. For details on these launch configuration types, refer to **http://help.eclipse.org**.

Wind River has added the **Remote Java Launch and Connect** launch configuration type which is documented here. This type of launch uses the usermode agent running on the remote target to start the application and then connect the debugger to the application. **Remote Java Launch and Connect** makes it easy to start debugging remote Java applications with a single click, and it also makes application I/O available in the Console view. This type of launch is required to debug combined Java and JNI code as described in *18.5 Java-JNI Cooperative Debugging*, p.302.

Create a Java Project

Create new Java projects in Workbench by selecting **New > Project > Java Project**, and import existing **.java** files or use the Editor to create new ones. Build the

project by right-clicking the project and selecting **Build Project**. A successful build generates class files and reports no errors. The following discussion assumes you are able to successfully build a Java application in Workbench and now wish to debug it on a target.

Prepare the Target

The target must have a supported Java runtime environment (see your *Release Notes* for details) and be running the Wind River usermode agent.

1. If your target does not have a **usermode-agent** executable on it, copy one to it.

   Source and binary files for **usermode-agent** may be found in your Workbench installation directory under the **linux-2.x/usermode-agent/2.0/** subdirectory.

NOTE: If you have both a **1.1** and **2.0** subdirectory, use the contents of the **2.0** directory for remote Java launches.

   If there is no binary available for your target, refer to *3.11 Working with External Locations*, p.84 for an example of a way to build **usermode-agent** from the source files with Workbench.

2. Run the usermode agent on the target, for example:

```
target_$ ./usermode-agent -p 4321 &
```

   In this case, a port is specified so that additional instances of **usermode-agent** can be run on the target for other purposes. (To set **usermode-agent** to run when the target boots, see *G. Starting usermode-agent at Boot Time*.)

NOTE: If you specify a port when invoking **usermode-agent**, you must also specify the same port number when you create the target connection from the host. This should be a unique number less than 65536 and not used as a port by any other application. Also, it should be greater than 1024, which is the last of the reserved port numbers.

3. Make the Java application class files available for execution on the target. For example, you could build the application in a Workbench project that is on a filesystem shared between the host and target, or simply copy the class files from the host to the target.

4. Make a target connection or create a new target connection to your target in the Remote Systems view. If you specified a specific port when starting **usermode-agent** on the target, you must specify that port in the target connection dialog as well. (For details on how to create a target connection, refer to *13. Connecting to Targets*.)

Launch the Remote Debugging Session

The following procedure provides an example of how to initiate a remote Java launch and debugging session. Only some of the possible launch configuration settings are discussed. For details on all launch configuration settings, refer to *Wind River Workbench User Interface Reference: Launch Configuration Dialog*.

1. Select **Run > Debug** and then select the **Remote Java Launch and Connect** configuration type. Click **New** to open a set of tabs. Tabs marked with a red X require input from you.
2. In the **Main** tab:
   a. Specify a name for your launch in the **Name** field or leave the default name.
   b. Click **Browse** to find the correct project and click **Search** to identify the correct main class.
   c. Check **Stop in main** to cause the debugger to suspend execution on entry into the method **main** in the main class.
3. In the **Connection** tab:
   a. Select the correct target connection from the **Connection to use** pull-down menu.
   b. If you are using an NFS mount and have not specified object path mappings for your target connection, you can do that now by clicking **Properties**, select the **Object Path mappings** tab, and click **Add**. Enter the host paths and target paths that map. For example, if you have mounted your target's root filesystem on the host's **/target** directory, enter **/** for the target path and **/target** for the host path. Click **OK** then click **OK** again to return to the launch configuration tabs.
   c. Enter or browse to the path of the **java** binary on the target in the **Exec Path on Target** field. For example, if the **java** binary on the target resides in the **/usr/bin** directory, enter **/usr/bin/java**.
4. In the **Arguments** tab, enter the directory path to the class files in the **Classpath** field. For example, if the target path **/java/classfiles/** holds the class files for your Java application, enter **/java/classfiles/** in this field.

   Note that the complete command line is displayed at the bottom of this tab.
5. In the **Debug Options** tab, enter a port number. This is the debugging port number for JDT—not the usermode agent port number. This should be a

unique number less than 65536 not used as a port by any other application, and it should be greater than 1024 which is the last of the reserved port numbers.

6. Click **Debug**.

   Workbench changes to the Debug perspective, and the Debug view shows the status of the launch. If you checked **Stop in main** in the **Main** tab of the launch configuration, you should see the application suspended at main. The source file should also be open in an Editor window, with the location where execution has been suspended highlighted.

   At this point you can examine variables, set breakpoints, step through your application, and so on. Output and standard error output will appear in the Console view, which is discussed in *17.6 Launches and the Console View*, p.281.

   Click the **Resume** icon in the Debug view toolbar to continue program execution.

17.4 Launching Programs Manually

Once a launch configuration has been established, you can run programs using the **Run** or **Target** selections on the menu bar, or using the **Run** and **Debug** buttons on the toolbar. Workbench will automatically try to connect to the target, if a connection is not already running.

Workbench checks the executable file running as a Linux process on the target against the host counterpart present on the host and accessible by the host's debugger server. These executables may be the same, if both the target and the host have access to the same file system. But they can differ, too. Workbench pops up a warning if it finds differences in their section sizes (such as in the **.text**, **.bss** and **.data** sections). This might warn you, for instance, that you have rebuilt the host executable, but you have forgotten to update the executable running on the target.

(The target executable must have the same sections, but it need not have any debug information, if, for example, the target has limited resources.)

17.5 Controlling Multiple Launches

You can create a Launch Control launch, consisting of a sequence of your launch configurations, each one of which is then considered a sub-launch. You can even add other Launch Control launches to a Launch Control configuration, the only restriction being that a Launch Control configuration cannot contain itself.

For detailed information on launch control settings, see the *Wind River Workbench User Interface Reference: Launch Configuration Dialog*.

Terminology

A *launch* is a specific instance of a launch configuration, and a launch configuration is a specific instance of a launch type. The launch is what occurs when you initiate a run or debug session.

A *launch configuration* is your definition of how the launch will occur, for example, what program will be run, what target it will run on, and what the arguments are.

A *launch type* defines the kind of launches that are supported by Workbench. There are several different kinds of launch types, for example, Java Application, Process on Target, and Launch Control. The launch type includes GUI elements that specify attributes specific to it.

You create a launch configuration based on a launch type, specifying the appropriate attribute values. You then initiate a launch based on a launch configuration. Launches also have a *mode*, the two standard modes being Run and Debug. A launch may be initiated by the **Run** or **Debug** buttons in Workbench (launches may be initiated other ways too). Note that some launch types are only available in one mode. For example, the **Remote Java Application** launch type can only be used in Debug mode.

Configuring a Launch Sequence

The following procedure assumes you have two or more launch configurations already defined (see *17.2 Creating a Launch Configuration*, p.266).

1. Select **Run > Debug** and the Debug dialog opens.
2. Select **Launch Control** from the **Configurations** list on the left, and then click **New** at the bottom. A new launch control configuration with the default name **New Configuration** appears. Change the name as desired.

3. Select the **Launch Control** tab. Note that your current launch configurations are listed under **Available Configurations** on the left, and a space on the right is labeled **Configurations to Launch**.
4. Select each launch that you want to add to your new launch configuration and click **Add** to add it to the list of configurations to launch. When you have a list of configurations to launch, you can organize them in the order you want them to launch by selecting a configuration and clicking **Move Up** or **Move Down**. The sub-launch at the top of the list will come first and the one at the bottom last. You can remove any sub-launch from the Launch Control configuration by selecting it and clicking **Remove**.

You now have a Launch Control configuration that will launch a sequence of sub-launches in the order specified in the **Configurations to Launch** list. You can also specify commands to perform before launches, after launches, and in response to a launch failure or an application error report as discussed in the next section.

Each launch in a Launch Control will open a Console view for I/O and error messages as described in *17.6 Launches and the Console View*, p.281.

Pre-Launch, Post-Launch, and Error Condition Commands

To access the launch configuration commands, select a sub-launch in your **Configurations to Launch** list and click **Properties** (or double-click the sub-launch). A properties page containing command information is displayed. Here you can specify pre-launch, post-launch, and error condition commands, which will inherit the environment variables shown below them unless you change them in the command. Your changes affect the launch you are working with only—other launches using the same configuration get the default values for the environment variables. Also, the set of environment variables differs for each launch configuration (see *Understanding the Command Environment*, p.279 for more on environment variables).

Preparing a Launch with a Pre-Launch Command

An example of the use of a pre-launch command is to prepare a target for use. For example, in a development environment you might have to reserve a target, and you would not want to attempt a launch without being sure you had a target to launch on. So a pre-launch command might be a script that reserves the board, puts **usermode-agent** in the root file system, reboots the board, and starts **usermode-agent**.

If the pre-launch command returns a non-zero return code then the launch is aborted and the error condition command is executed for each sub-launch previous to the failed sub-launch.

Using a Post-Launch Command

If your application requires additional set up after it has been launched, or if you would like to verify that it has launched correctly before proceeding to the next launch, use a post-launch command.

If the post-launch command returns a non-zero return code then the launch is aborted and the error condition command is executed for each sub-launch previous to the failed sub-launch as well as for the failed sub-launch.

Using the Error Condition Command

The error condition command of a launch is run when a launch fails, or a pre-launch or post-launch command returns a non-zero error code. This causes the error command of the current launch to run, and then each error command of any preceding launches to run. The error condition commands are executed in reverse order of the sequence in which the launches occurred. For example, if the fourth launch fails, the error condition command of the fourth launch is performed, then the error condition of the third launch, and so on. This is to deal with situations in which previous commands may have acquired locked resources—unlocking them in reverse order is important to prevent potential deadlock.

NOTE: To be precise, error commands are called in the reverse order that the pre-launch commands were called. An error command will never be called for a sub-launch that did not pass the pre-launch command step.

Inserting Commands using an Empty Sub-Launch

You can place a command into your Launch Control that is not associated with any particular sub-launch by adding an empty Launch Control to hold the command. Select Launch Control and click **New** and then specify a name for the dummy launch, for example, **Empty Launch**. Add the empty launch to the Launch Control and use the properties page to insert commands into the launch which are not associated with any particular sub-launch.

Running All Pre-Launch Commands First

If you want to run each of the pre-launch commands for each launch first, check **Run Pre-Launch command for all launches first** on the main launch control page. The pre-launch commands will be executed in order, and only after they are all

successfully completed will the first launch take place, followed by the second launch and so on. This provides for situations in which you do not want to continue with a complete launch Control sequence if any of the sub-launches cannot take place because, for example, a target is not available.

Launch Controls as Sub-Launches

You can use an existing Launch Control as a sub-launch, but do not attempt to create recursive launch controls in this way, as they will not run.

If the parent Launch Control's **Run Pre-Launch command for all launches first** is selected and the pre-initialize check box is set for the child Launch Control, the child will pre-initialize all of its sub-launches before operation continues on to the next sub-launch of the parent Launch Control. Otherwise, the child Launch Control will have its sub-launches initialize at the time that it is launched.

Understanding the Command Environment

The environment variables are collected from multiple locations and then provided on the Properties page as a convenience. Typically you will only read variable values, but you may want to change them in your pre-launch command. Your changes affect the launch you are working with only—other launches using the same configuration get the default values for the environment variables.

Environment variables are gathered from four different sources:

- First, variables may be defined on the Launch Control's Environment tab. These variables are not displayed on a sub-launch's Properties page because the information is readily available on the Environment tab.
- The second source is from the sub-launch's Environment tab (if it has one).
- The third source for the list of environment variables is defined by the sub-launch's configuration type attributes. Each sub-launch configuration type defines its own set of attributes: see the Eclipse documentation for Launch Configuration for details on sub-launch attributes.
- The fourth source of environment variables are defined by Launch Control and provide general support for the launch. The variables defined by Launch Control for each sub-launch are:
  - **com_windriver_ide_launchcontrol_launch_mode**
  - **com_windriver_ide_launchcontrol_env_file**
  - **com_windriver_ide_launchcontrol_skip_next**

The environment variable **com_windriver_ide_launchcontrol_launch_mode** identifies the mode of a launch. The mode may be either **debug** or **run**, depending on how a launch is initiated (for example selecting the **Run > Debug** dialog to initiate a debug mode launch and **Run- > Run** to initiate a run mode launch). Changing **com_windriver_ide_launchcontrol_launch_mode** has no effect—it is only provided for information about a current launch.

Since the command's environment terminates after the command completes any variables which need to be changed for a launch must be written to a file. The name of this file is provided in the environment variable **com_windriver_ide_launchcontrol_env_file**. The format of this file is a list of key value pairs on separate lines. Each key and value is separated by an = and the key identifies the variable name (this is a standard Java properties file). After a command is completed Launch Control will read this file and update any variables as specified in the file.

Launch control also defines the **com_windriver_ide_launchcontrol_skip_next** variable. Setting this variable to **true** in the Pre-Launch command causes the remainder of the sub-launch to be skipped. Setting this variable in post-launch or error commands has no effect.

An example of how this could be used is to check for the existence of a server application in a pre-launch command. If the application is already running then specifying **com_windriver_ide_launchcontrol_skip_next=true** in the **com_windriver_ide_launchcontrol_env_file** will cause the launch of the application to be skipped without invoking an error.

NOTE: Note that the Wind River environment variables for individual launches are subject to change and you should not count on them being maintained across releases. For details on variables beginning with the string **org_eclipse** refer to the documentation available at **http://help.eclipse.org**.

17.6 Launches and the Console View

Workbench supports the Eclipse **Console** view with Virtual IO (VIO) features that allow you to monitor the standard output and error output of your applications and to enter standard input. VIO connects the **Console** view to a particular context (process or task). You can also have multiple **Console** views and "pin" them to a particular context. Most **Console** view settings are available in the **Common** tab of your launch configuration, and you can specify **Console** view preferences in your Workbench preferences.

Note that **Console** view VIO is tied to the debugger and cannot always serve the same purposes as a terminal connection to the target. You cannot use it, for example, to monitor the boot loader or set boot parameters. The **Console** view is associated with a particular debugger context and is not a general purpose terminal connection.

Launches and the Console View

Each launch opens a Console view for I/O and error messages, provided the **Allocate Console** check box is selected in the Common tab of the launch (the default setting).

NOTE: This refers to the Common tab of each individual launch configuration, not the Common tab of the Launch Control configuration.

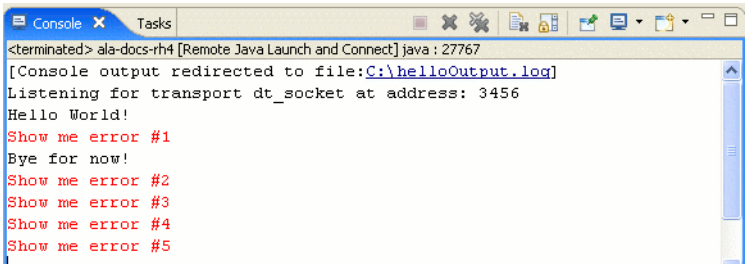
In the Common tab you can also specify a file where console output is appended or overwritten with each launch. The Console view itself offers several controls as described in the next section.

Note that you can also modify Console view settings such as buffer size and text colors by selecting your preferences at **Window > Preferences > Run/Debug > Console**.

Console View Output

To open a **Console** view select **Window > Show View > Other > Basic > Console**. An example view is shown below.

Figure 17-2 **Example Console View**


Console
Tasks
<terminated> ala-docs-rh4 [Remote Java Launch and Connect] java : 27767
[Console output redirected to file:C:\helloOutput.log]
Listening for transport dt_socket at address: 3456
Hello World!
Show me error #1
Bye for now!
Show me error #2
Show me error #3
Show me error #4
Show me error #5

The highlights of the view shown include the following:

- A title indicates which context (process or task) this view applies to.
- A comment indicates that in this case console file logging is occurring and identifies the log file location. click the filename to display it in the Editor.
- The standard output shown in the example is **Hello World!** and **Bye for now!** and is in black, the default color for standard output.
- The standard error outputs shown in the example are the **Show me error** messages which are in red, the default color for standard error output.

NOTE: The output appearing in the Console view can appear in a different order than the order the output was produced if both output and error output are present. The data from these two output types go through different channels and their transit times can be different.

Along with other standard functions, icons in the **Console** view toolbar allow you to pin the context to a Console view, select among different Console views, and create new Console views.

Select a specific process or task for a Console view by clicking the down arrow next to the **Display Selected Console** icon and making your selection. Click **Pin Console** to keep the Console view associated with that context. Select **Open Console > New Console View** to create additional Console views.

Refer to **http://help.eclipse.org** for further details on the Console view, or press **CTRL-F1** in the Console view for online help.

17.7 Attaching the Debugger to a Running Process

You can attach to a running process to debug it as follows:

1. In the Remote Systems view, expand **Processes** for the target connection and locate the process you want to attach to.
2. Right-click the process and select **Attach to Process**.

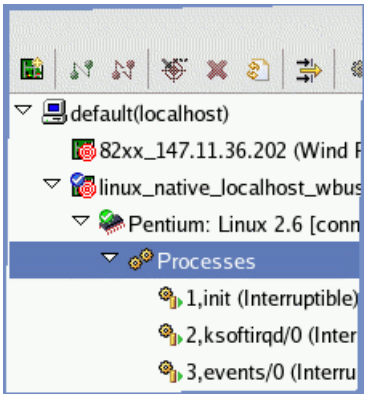
If you compiled the process with debug symbols, the symbols should load to allow source-level debugging.

Right-clicking a process executable from the Project Explorer and selecting **Run Process on Target** or **Debug Process on Target** opens the appropriate launch configuration dialog box. See *17.2 Creating a Launch Configuration*, p.266 for more information about working with these dialog boxes.

This section explains how to launch processes from the Remote Systems view.

1. Right-click **Processes**, then select **Run/Debug Process**. The **Run Process** dialog box appears.
2. Type the path and filename (as seen by the target) into the **Exec Path on Target** field, or click the **Browse** button and navigate to the executable file.
3. To immediately put the program under debugger control at launch, select **Attach Debugger** and **Break on entry**; to let it run, clear the **Break on entry** check box. Click **OK**.

   Workbench runs the process on the target, and the executable and its host location, along with the individual tasks, appear below **Processes** in the Remote Systems view. If a red **S** appears, then symbol information has been loaded into the debugger.



If you selected **Break on entry at main**, four other things happen as well:

- Workbench automatically switches to the Device Debug perspective (if it is not already open).
- The process is displayed in the Debug view.
- A temporary breakpoint is planted and appears in the Breakpoints view.
- The program executes up to the entry point at **main** and breaks.

Whenever you manually run a process, a corresponding **Attach to Target** launch configuration is automatically created. To use these configurations, see *Editing an Attach to Target Launch Configuration*, p.267.

17.8 Attaching to the Kernel

You can attach to the kernel in KGDB debugging mode.

For an example of attaching to core with a KGDB connection, refer to *4.5 Attaching and Debugging the Kernel from Workbench*, p.99.

17.9 Suggested Workflow

Launch Configurations allow for a very tight Edit-Compile-Debug cycle when you need to repeatedly change your code, build and run it. You can use the **F11** (Debug Last Launched) key to build the projects you have specified, connect your target (unless it is already connected), download, and run your most important program over and over again.

The only thing to watch is that you cannot rebuild your program or kernel while it is still being debugged (or its debug info is still loaded into the debugger). Depending on the size of the modules you run and debug, it can be the case that the debug server cannot load all the symbolic information for your modules into memory. By default, the size limit is set to 60MB (this can be changed by selecting **Window > Preferences > Remote Systems > Debug Server Settings > Symbol File Handling Settings**.)

If a module is bigger than this limit, it will be locked against overwriting as long as the debugger has symbols loaded. This means that when you try to rebuild this module, you will see a dialog box asking you to unload the module's symbol information from the debugger before you continue building. You can usually unload symbolic information without problems, provided that you do not have a debug session open in the affected module. If you have a module open, you should terminate your debug session before continuing the new build and launch process.

18 Debugging Projects

18.1 Introduction 287
18.2 Using the Debug View 288
18.4 Using the Disassembly View 301
18.5 Java-JNI Cooperative Debugging 302
18.6 Remote Kernel Metrics 306
18.7 Run/Debug Preferences 308

18.1 Introduction

The Wind River Workbench debugger lets you download object modules, launch new processes, and take control of processes running on the target. Unlike other debuggers, you can attach to multiple processes simultaneously, without affecting them or being required to disconnect from one process in order to attach to another.

This chapter introduces the Debug and Disassembly views, and shows you how to use them to debug your programs.

NOTE: You must use the **-g** compiler option to use many debugger features. The compiler settings used by the Workbench project facility's Managed Builds include debugging symbols.

18.2 Using the Debug View

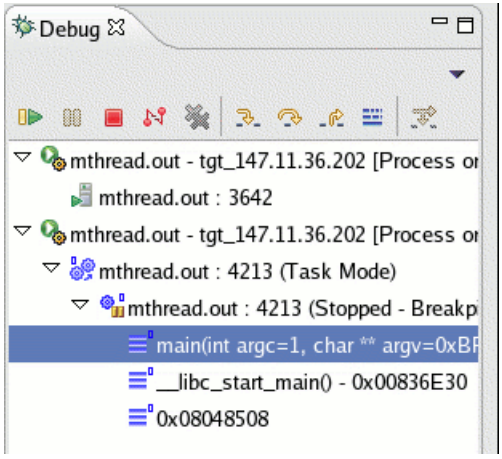
Use the Debug view to monitor and manipulate the processes running on your target.

To examine a process in the Debug view:

1. Connect to your target in the Remote Systems view (see *13.4 Establishing a Connection*, p.230).
2. Launch one or more processes:
   - Using a launch configuration as described in *17.2 Creating a Launch Configuration*, p.266.
   - Manually, as described in *17.4 Launching Programs Manually*, p.275.
   - By attaching to an already running process, as described in *17.7 Attaching the Debugger to a Running Process*, p.283.
3. Once the debugger has attached to your process, it will appear in the Debug view.

The Debug view displays processes differently depending on whether the debugger is attached or not. Figure 18-1 shows two instances of the **mthread** process: the first instance does not have the debugger attached and the second does.

Figure 18-1 **Debug View with Unattached and Attached Processes**



18.2.1 Configuring Debug Settings for a Custom Editor

By default, the Workbench Editor opens when the debugger stops in a given file. To cause a different editor to open for particular file types, modify the mappings in **Window > Preferences > General > Editor > File Associations**.

Modifying these mappings takes care of editor selection and painting of the instruction pointer in the editor gutter. However, to associate other debugging actions with the new editor, you must modify the Eclipse extension point **org.eclipse.ui.editorActions**.

For example, the breakpoint double-click action associated with the Workbench Editor looks like this:

```
<extension point="org.eclipse.ui.editorActions">
   <editorContribution
       targetID="com.windriver.ide.editor.c"
       id="com.windriver.ide.debug.CSourceFileEditor.BreakpointRulerActions">
     <action
       label="Dummy.label"
       class="com.windriver.ide.debug.internal.ui.breakpoints.actions.ToggleB
reakpointRulerAction"
       actionID="RulerDoubleClick"
       id="com.windriver.ide.debug.ui.actions.toggleBreakpointRulerAction.c">
     </action>
</editorContribution>
```

Other features that are by default configured to work only with the Workbench Editor are **Run to line**, **Set PC to here**, and **Watch**. These features are configured through following extensions:

```
<viewerContribution
      targetID="#WREditorContext"
      id="com.windriver.ide.debug.ui.editprpop-up.actions">
   <visibility>
      <and>
         <systemProperty
            name="com.windriver.ide.debug.ui.debuggerActive"
            value="true"/>
         <pluginState value="activated" id="com.windriver.ide.debug.ui"/>
      </and>
   </visibility>
   <action
        label="%WatchAction.label"
        icon="icons/actions/hover/watch_exp.gif"
        menubarPath="group.debug"
        helpContextId="com.windriver.ide.debug.ui.watchAction_context"
        class="com.windriver.ide.debug.internal.ui.actions.WatchAction"
        id="com.windriver.ide.debug.ui.editor.watchAction">
      <enablement>
        <systemProperty
           name="com.windriver.ide.debug.ui.debuggerActive"
```

```
            value="true">
          </systemProperty>
        </enablement>
    </action>
    <action
          label="%SetPcToHereAction.label"
          menubarPath="group.debug"
          helpContextId="com.windriver.ide.debug.ui.setPcToHereAction_context"
          class="com.windriver.ide.debug.internal.ui.actions.SetPcToHereAction"
          id="com.windriver.ide.debug.ui.editor.setPcToHereAction">
    </action>
    <action
          label="%RunToLineAction.label"
          icon="icons/actions/hover/run_to_line.gif"
          menubarPath="group.debug"
          helpContextId="com.windriver.ide.debug.ui.runToLineAction_context"
          definitionId="org.eclipse.debug.ui.commands.RunToLine"
          class="org.eclipse.debug.ui.actions.RunToLineActionDelegate"
          id="com.windriver.ide.debug.ui.editor.runToLineAction">
    </action>
</viewerContribution>
```

Refer to Eclipse SDK documentation for more about these extension points.

18.2.2 Understanding the Debug View Display

The Debug view displays a hierarchical tree for each process being debugged. The following examples show what might appear at each level of the tree, with a general description of each level.

ball (2) [Process on Target] = launch level
launch name [launch type]

> **16,ball.out (MPC8260: Linux 2.6)** = debug target level
> *process name (core name:OS name OS version)*
>
> > **16,ball.out (Stopped - Breakpoint)** = thread level
> > *thread name (state - reason for state change)*
> >
> > > **main( ) - main.c:59** = stack frame level
> > > *function(args) - file : line #*, can also be *address*

Stack arguments and argument values are not displayed in the Debug view by default, so as to improve debugging performance.

To activate stack-level arguments in the Debug view, select **Window > Preferences > Run/Debug > Performance**, then select the **Retrieve stack arguments for stack frames in Debug View** and **Retrieve stack argument values for stack frames in Debug View** check boxes. Click **OK**.

NOTE: The stack arguments reflect the current values of the stack argument variables, not their initial values immediately after entering the function call.

How the Selection in the Debug View Affects Activities

Choosing a specific level of your debug target controls what you can do with it.

Selected Level	Action Allowed
launch	Terminate or disconnect from all processes/cores for the launch debug target.
debug target	Terminate or disconnect from the debug target.
	Perform run control that applies to the whole process: suspend/resume all threads.
	Assign color to the debug target and all its threads/tasks.
thread	Terminate or disconnect; terminates individual tasks/threads, if supported by process/core.
	Run control for thread: resume/suspend/step.
	Assign color to thread.
stack frame	Select of the stack frame causes the editor to display instruction pointer and source for stack frame.
	Perform same run control as on the thread.
	Assign color to thread.
	Assign corresponding color for parent thread.

Monitoring Multiple Processes

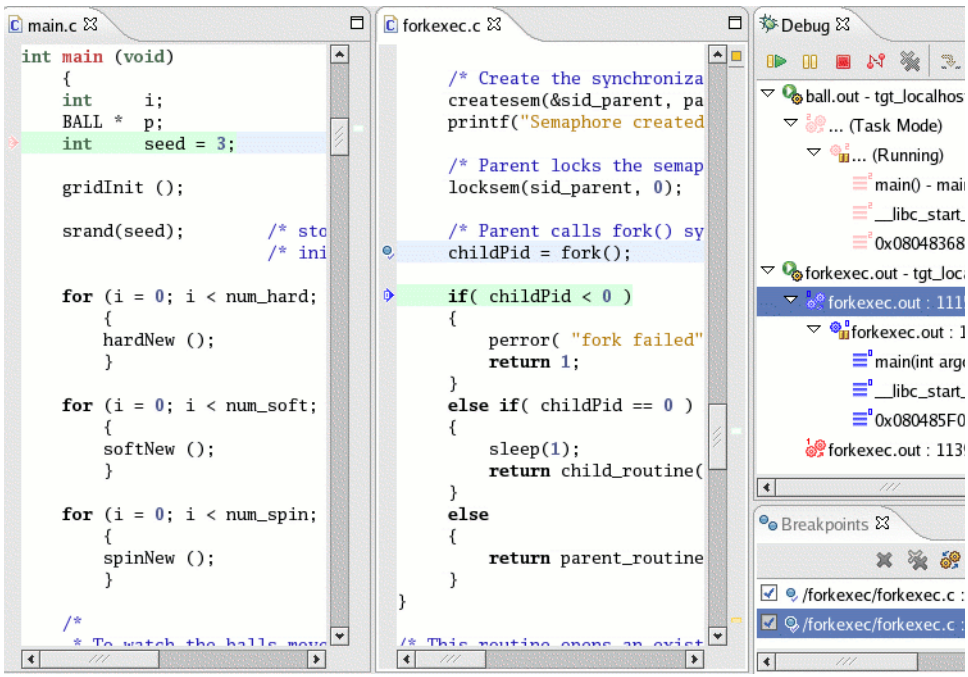
When you start processes under debugger control, or attach the debugger to running processes, they appear in the Debug view labeled with unique colors and numbers. Likewise, breakpoints that are restricted to a particular process display that process's color/number context in the Breakpoints and Editor views.

For example, in Figure 18-2, three processes are shown in the Debug view:

- The **ball** process, in pink in the Debug view, has been launched in debug mode and the program counter is shown, in pink, in the **main( )** routine.
- The **forkexec** process is shown in blue. It has stopped at a breakpoint set at the **fork** system call. The breakpoint is shown as a solid circle and the program pointer is shown in blue with the number 0 in it. Note that the number 0 is also shown with the parent process in the Debug view.
- The third process, the forked child process, is shown in red in the Debug view.

The color assigned to a process or thread can be changed by right-clicking the process or thread and selecting **Color >** *specific color*.

Figure 18-2 **Debug View with Editor and Breakpoint View**



The context pointer (the arrow in the left gutter in **main.c**) indicates the statement that will execute when the process resumes.

Colored Views

The color context of a process also carries through to other views whose scope is determined by the contents of the Debug view.

The data views that appear in the Device Debug perspective usually update to reflect whatever is currently selected in the Debug view. If you prefer, you can start colored views that are *pinned* to a process of a particular color and update only when that process changes.

To open a view of a particular color, select **Window > Show View > Other > Device Debug -** *color* **>** *view.*

For more information about how to set up Debug view settings, see the *Wind River Workbench User Interface Reference: Debug View.*

18.2.3 Stepping Through a Program

Once a process has stopped under debugger control (most often, at a breakpoint), you can single-step through the code, jump over subroutine calls, or resume execution; what you can do depends on what you selected in the Debug view.

When the program is stopped, you can resume operation by clicking **Resume** on the toolbar of the Debug view. If there are no more remaining breakpoints, interrupts, or signals, the program will run to completion (unless you click the **Suspend** button).

To step through the code one line at a time, click the Debug view's **Step Into** button. If you have other data views open, such as the Registers or Variables views, they will update with current values as you step through the code.

The effect of **Step Into** is somewhat different if you click **Toggle Disassembly/ Instruction Step Mode** on the Debug view toolbar, or when the current routine has no debugging information. When this mode is set, the step buttons cause instruction-level steps to be executed instead of source-level steps. Also, the Disassembly view will be shown instead of the Editor.

To single-step without going into other subroutines, click **Step Over** instead of **Step Into**.

While stepping through a program, you may conclude that the problem you are interested in lies in the current subroutine's caller, rather than at the stack level where your process is suspended. Click the Debug view's **Step Return** button in that situation: execution continues until the current subroutine completes, then the debugger regains control in the calling statement.

The **Run > Stack Frame** menu provides other options for manipulating files:

Drop To Frame

The debugger resumes until the execution returns to the selected stack frame.

Run To Frame Address

The debugger executes until reaching the address of the selected stack frame.

Set Breakpoint at Frame Function

Create an expression breakpoint at *Frame Function* of the selected stack frame (the actual function name replaces *Frame Function*). This action is only available if the symbol data for the selected frame is present (and the function name is known).

Set PC to Frame Function

Set the Program Counter register to the beginning of the *Frame Function* of the selected stack frame, with *Frame Function* replaced by an actual function name. Only available if the symbol data for the selected frame is present (and the function name is known).

18.2.4 Using Debug Modes

Depending on the type of connection the debugger has to the target, the debugger may be capable of operating in different modes. Different debug modes have different capabilities and limitations, which are mostly related to how the debugger interacts with the target and the processes that are being debugged.

Your target connection type determines possible modes as follows:

- KGDB connection type – Only supports debugging the kernel using a single execution context. When the system context is suspended, the kernel, kernel threads, and user processes are suspended also.
- Usermode agent connection type – Supports debugging user processes. Processes and threads within processes are suspended and resumed independently of each other.

As a general rule, when the target is being debugged in user mode, the debugger interacts only with the process or processes being debugged. If this process is suspended, other processes keep running. This mode is less intrusive, as it allows the user to control the selected process or thread while the rest of the system can continue to operate normally.

In kernel mode, the debugger interacts with the entire system at once, so if one task is suspended, all processes and kernel tasks running on the system are suspended as well. This allows for increased control of and visibility into what is happening on the system, but it is also very disruptive.

For example, if the system maintains network connections with other systems, suspending it will cause the others to lose their network connections with the debugged system.

18.2.5 Setting and Recognizing the Debug Mode of a Connection

Right-clicking a connection in the Remote Systems or the Debug views and selecting **Target Mode** allows you to specify a debug mode for the connection. The currently active mode is indicated by a checkmark.

When you create a new debug connection through a launch, the connection debug mode (either system mode or task mode) is saved as a property of the launch. This mode is listed in parentheses at the end of the label of the target node in the Debug view.

For target connections that support switching between modes, if you switch the debug mode while a debug connection is active, this debug connection will become unavailable in the Debug view. When a debug connection is unavailable, no operations can be performed on it, except for disconnecting the debug connection.

In the Remote Systems view, if you switch the target to system mode, every node in the tree will have a system mode icon painted on top. If the system mode icon does not appear, then the node and processes are in task or user mode.

18.2.6 Debugging Multiple Target Connections

You can debug processes on the same target using multiple target connections simultaneously. An example of this setup is a Linux target that has a user mode **ptrace** agent installed for debugging processes, and an OCD connection for halting the system and debugging the kernel.

In this situation, if the system is halted using the OCD (system mode) target connection, the user mode **ptrace** agent will also be halted, and the user mode target connection will be lost. When the system is resumed, the user mode target connection will be re-established.

The Remote Systems and the Debug views (if a debug session is active) both provide feedback in this scenario. The Remote Systems view hides all the process information that was visible for the target, and displays a label **back-end connection lost** next to the target node. The Debug view does not end the active debug session, but it shows it as being **unavailable**, in the same manner as if the debug mode was switched.

18.2.7 Disconnecting and Terminating Processes

Disconnecting from a process or core detaches the debugger, but leaves the process or core in its current state.

Terminating a process actually kills the process on the target.

NOTE: If the selected target supports terminating individual threads, you can select a thread and terminate only that thread.

18.2.8 Using the Debug View to Send Signals to Processes

In POSIX systems, a *signal* is an asynchronous notification sent to a process that an event has occurred. Sending a signal to a process interrupts its normal execution.

The Debug view is aware of signals sent to processes, and displays information about them. If a process is suspended because of a signal, the view displays the signal name and description. For example, if a process is suspended because it receives a SIGHUP signal, the Debug view shows the information **(Stopped - Signal Received: SIGHUP, Hangup)**.

Resuming Suspended Processes

You can resume a suspended process with or without a signal.

To resume a process without a signal:

1. Right-click in the Debug view.
2. Select **Resume Without Signal**.

   The selected process resumes, discarding any pending signal.

To resume a process with a signal:

1. Right-click in the Debug view.
2. Select **Resume With Signal**.

   This opens a dialog from which you can select a signal and click **OK**. The selected process resumes, and immediately receives the selected signal.

Editing a Process's Signal Configuration

Each signal can be configured to be passed to the process, or to force the process to suspend when it receives the signal.

To edit the signal configuration of any process:

1. Right-click the process in the **Debug** view.
2. Select **Edit Signal Configuration**.

   This opens a dialog allowing you to configure signal attributes.
3. Click a value to change it from **true** to **false**, or from **false** to **true**.

The **Pass** attribute controls whether a signal sent to a process should actually be passed to it. This attribute's values act as follows:

- When **true**, the process silently discards the signal.
- When **false**, the process cannot silently discard the signal.

The **Suspend** attribute controls whether the process should suspend or not when receiving the signal. This attribute's values act as follows:

- When **true**, the signal suspends the process when the process receives it.
- When **false**, the process continues running after it receives the signal.

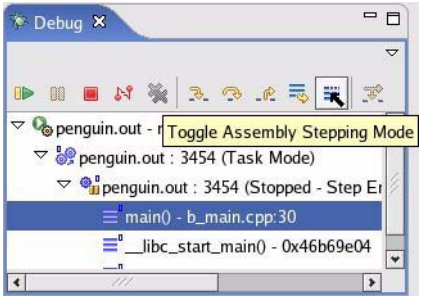
18.2.9 Changing Source Lookup Settings

Source Lookup maps debugger source file paths to the actual file locations in the workspace and the host file system. The paths for source files are read by the debugger (from the symbol data of the executable being debugged). The compiler generated them when the executable was built, and they are often different from the paths to those files when debugging. For information about how to set up Source Lookup Path settings, see the *Wind River Workbench User Interface Reference: Source Lookup Path Dialog*.

18.3 Stepping Through Assembly Code

To view assembly code interleaved with the corresponding source code:

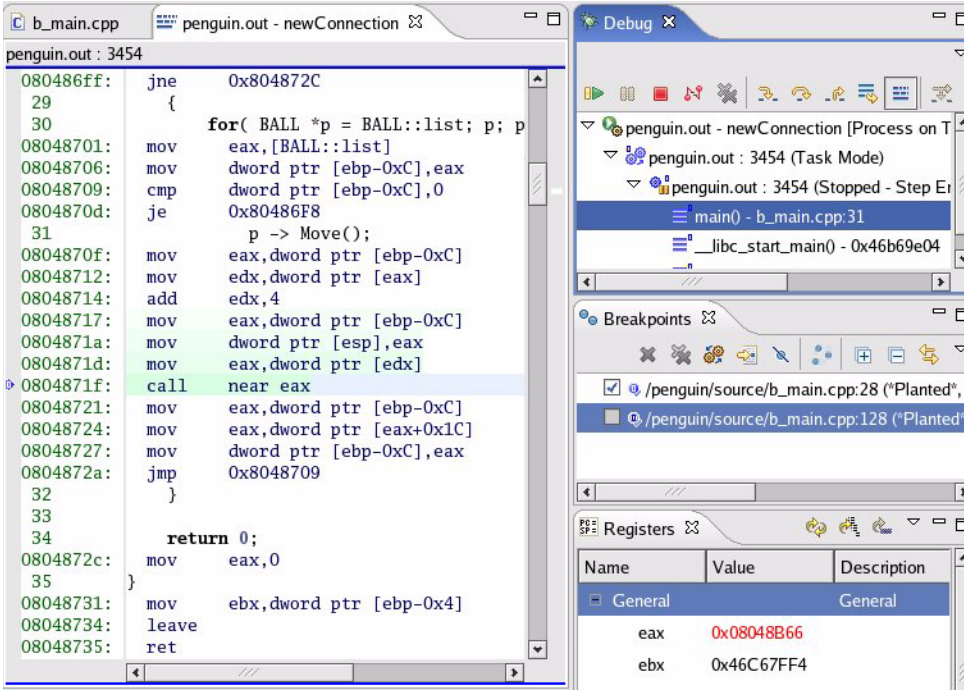
1. Click the **Toggle Assembly Stepping Mode** button as shown in the following example, which shows penguin in the Debug view:



2. Step or proceed to a breakpoint such as shown in Figure 18-3 as a mix of assembly and C++ source code.

   In this case, the code has been single-stepped to the **call** instruction. Note that the previous steps are shown in lessening degrees of shading—this helps you see where you've been when you step into and out of routines.

Figure 18-3 **Single Stepping Though Assembly Code**

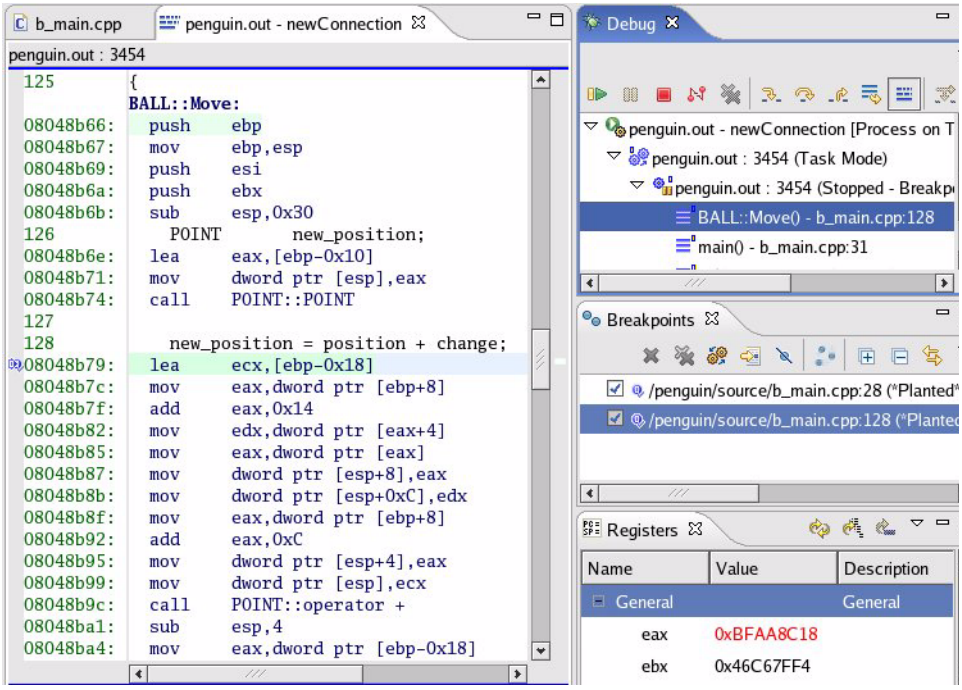


NOTE: The assembly code shown will differ depending on your target. Code produced for an Intel MPCBL0001 target is shown. For the ARM Versatile board, for example, the **blx** instruction corresponds to the **call** instruction.

With the assembly language code visible, you can step in and out of the inherited parent class methods for C++ classes as your C++ code is executing.

For example, if you step into the call for the **p** object's **Move** method, you step into the **Move** method code for the parent class **BALL**, as shown in Figure 18-4.

Figure 18-4 **Example of C++ Assembly Code**



Also you can see, at line 126, the **call** to the parent class constructor **POINT::POINT**, which creates a new **POINT** object for the local variable **new_position**. Additionally, you can see the call (for line 128) to the parent class operator method **POINT::operator +**.

NOTE: This is all hidden when you debug just at the source code level.

18.4 Using the Disassembly View

Use the Disassembly view for the following purposes:

- To examine a program when you do not have full source code for it (such as when your code calls external libraries).
- To examine a program that was compiled without debug information.
- When you suspect that your compiler is generating bad code (the view displays exactly what the compiler generated for each block of code).

18.4.1 Opening the Disassembly View

Unlike other Workbench views, the Disassembly view is not accessible from the **Window > Show View** menu—it appears automatically if the Debug view cannot display the appropriate source code file in the Editor (it appears as a tab in the Editor, labelled with the target connection being debugged).

The Disassembly view can be opened manually by clicking the Debug view's **Toggle Disassembly/Instruction Step Mode** toolbar button, and by right-clicking in the Stack Trace view and selecting **Go To Code**.

18.4.2 Understanding the Disassembly View Display

The Disassembly view shows source code from your file (when available), interspersed with instructions generated by the compiler. As you step through your code, the Disassembly view keeps track of the last four instructions where the process was suspended. The current instruction is highlighted in the strongest color, with each previous step fading in color intensity.

If the Disassembly view displays a color band at the top and bottom (here, the band is blue), then it is pinned to the process with that color context in the Debug view; if no color band is displayed, then the view will update as you select different processes in the Debug view.

For more information, see *Wind River Workbench User Reference: Disassembly View*.

18.5 Java-JNI Cooperative Debugging

Java-JNI (Java native interface) cooperative debugging allows you to debug the Java side of your Java application with the JDT debugger, and the JNI side of your application with a C/C++ native debugger, simultaneously. You can use Java-JNI cooperative debugging when you have a Java project that has Java classes with native methods, and have built the associated native libraries with debug information.

In addition to your Java classes, the JRE, and the native libraries required to run your application on the target, you must have the special helper class and its associated library file on the target as well. These are provided by Wind River and are required for Workbench to perform Java-JNI cooperative debugging.

The helper class file and its associated native library are available in the directory *installDir***/linux2-x/usermode-agent/bin/wrjnidebug/**. Copy the contents of this directory (the **wrjnidebughelper.ja**r file and the **libwrjnidebughelper.so** file) to a directory on the target. You will need to refer to the location of these files when creating your launch configuration as described in *Creating a Launch Configuration for Cooperative Debugging*, p.303.

You must also install JDT, as explained in *20.6 Installing JDT for Third-Party Plug-ins and Debugging*, p.321.

Configuring a User Mode Connection for Cooperative Debugging

If you do not have a user mode connection to your target, create a new connection (see *3.4.1 Creating and Configuring QEMU Connections*, p.61 for details). Be sure to specify the port number if you started **usermode-agent** with a port number.

If your target's root file system is accessible to the host computer on which you are running Workbench, specify the root file system in the **Target Server Options** page under the Target Connection properties for your connection. Right-click your target connection in the Remote Systems view and select **Properties** to access this tab.

If, however, your target's root file system is not fully accessible to the host computer, then you need to manually add the mappings under **Object Path Mappings** in the Target Connection properties for your connection. Right-click your target connection and select **Properties** to access this tab. Be sure to configure

your object path mapping so that the native debugger can locate the object files for the following:

- The **java** executable (for example, **/usr/bin/java**)

  Without access to symbols from the java executable, the native debugger will not be able to detect any shared libraries that the Java application loads, and hence you will not be able to debug your JNI code.

- The Linux run time loader (**ld.so** or **ld-linux.so**, usually located in **/lib**)

  Without access to symbols of the runtime loader, the native debugger will not be able to automatically detect shared libraries, such as your JNI libraries, as the Java VM loads them. This will again prevent you from debugging your JNI code.

- The JNI libraries that are part of your application

  Without access to symbols from your JNI libraries, you will not be able to debug your JNI library even if the native debugger can detect that the library has been loaded.

Note that if none of the target file system is accessible, you may need to copy the necessary object files from the target to a location to which the host has access, and then specify that location in your object path mappings.

Creating a Launch Configuration for Cooperative Debugging

Use the following procedure to create a launch configuration for cooperative debugging. For details on all options available when creating a launch configuration, refer to *Wind River Workbench User Interface Reference Manual: Launch Configuration Dialog* online.

1. Select **Run > Debug**, select **Remote Java Launch and Connect**, and click **New** to create a new launch configuration.
2. In the **Main** tab:
   - Enter the name of your Java project for **Project** or select it by browsing to it.
   - Enter your main class for **Main class** or select **Search** to select it from a list.
   - Check **Stop in main** (optional, but will be used in this example).
3. In the **Connection** tab:
   - Select your target connection in the **Connection to Use** drop-down list.

- Specify the target path to the Java executable in **Exec Path on Target**, for example, **/usr/bin/java**.
- Click **Edit** in **Environment** and remove the **LD_BIND_NOW** entry. Click **Add** to add an entry with a **Name** of **LD_LIBRARY_PATH.** For **Value,** specify the target path to the directory or directories on the target where the JNI library (or libraries) that your application will call reside.

4. In the **Arguments** tab:
   - Under **Classpath**, enter the full target paths to both the directory where your Java classes for your application reside, and the full path of the **wrjnidebughelper.jar** file, located where you copied the contents of the **wrjnidebug** directory. Separate multiple paths by colons. For example:

     ```
     /usr/myapp:/usr/myapp/jni:/usr/agent/wrjnidebug/wrjnidebughelper.jar
     ```

     Note that these are the paths as visible to programs running on the target, not host paths.
5. In the **Debug Options** tab:
   - Specify a debug port for **Port Number**. This should be a unique number not used as a port by any other application including **usermode-agent**, and it should be greater than 1024 which is the last of the reserved port numbers, and less than 65536.
   - Check **Enable Debugger Cooperative Mode for JNI**.

Debugging In Java and Native Modes

Once you have created your launch configuration for cooperative debugging you can begin debugging and switch between Java and native modes as described in the following procedure.

1. Click **Debug** to start your application.

   Your application will come up stopped in its main class (if you selected **Stop in main** in the Main tab). In the Debug view you should see two hierarchical trees—one for the Java debugger (**Java HotSpot™ Client VM**) and one for the native debugger (**java**). Execute your Java application until it reaches a statement where there is a call to a native method and then click **Step Into**. It should step into the native method.

   If you did not configure the helper files location correctly, you will get an error message indicating that JNI transitions are disabled.

2. To return from JNI debugging to Java debugging, click **Resume** (not **Step Return**) to complete the step. Workbench will stop the Java thread and you can continue with debugging the Java side.
3. You can now move between the native and Java debuggers at any time and set breakpoints in either—as long as the Java debugger is not disabled (discussed next). For example, open any Java or native source file that is part of your application and double-click in the gutter to set a breakpoint.

Conditions that Disable the JDT Debugger

With Java-JNI cooperative debugging, the native debugger has control over the entire process including the JVM. Because of this, it is possible for the entire Java application including the JVM to be stopped by the native debugger. This will happen, for example, if you choose to manually suspend the entire process from the native debugger.

You can also set a native breakpoint in your JNI library that will stop the entire process. This can result in all the threads of the application, including the ones communicating with the JDT Java debugger, to be suspended. When this happens, the JDT debugger is not able to debug the Java side as it is unable to communicate with the JVM.

In Workbench, the Java debugger entries in the Debug View will display the message "**(debugger is disabled)(may be out of synch)**", indicating that the Java debugger is disabled and the Debug view display of Java information may no longer be accurate. In this condition, you cannot perform any Java debugging operations such as planting Java breakpoints, inspecting Java variables, and so on.

Re-Enabling the JDT Debugger

Click **Resume** to resume the entire process from the native debugger (not just one thread). The Java debugger will re-establish communication with the JVM and become enabled. You can now continue with Java-JNI cooperative debugging.

18.6 Remote Kernel Metrics

Remote kernel metrics (RKMs) are operating system signals (metrics) dynamically collected by the **rkm_monitor_linux** target agent. They can be displayed in real-time using the color features of the Workbench Data Monitor GUI.

The RKM monitor acquires its data from the **/proc** filesystem on the target. If you do not have a **/proc** filesystem on your target, you may simply need to mount it, or you may need to rebuild your kernel to include it.

To mount the **/proc** filesystem, use the **mount** command as follows:

```
# mount -t proc proc /proc
```

If your kernel does not have **/proc** support built-in, you must re-build and enable it in the **File system** configuration section of your kernel configuration tool.

Building and Running the RKM Monitor

The RKM monitor agent source code is supplied in a Workbench sample project.

To build the RKM monitor and then run it on your Linux target:

NOTE: The following procedure assumes the results of your project build are available on the target, for example by locating your workspace on a shared NFS mount. It also assumes you have created a connection to the target in the Remote Systems view and have specified the target-host root filesystem mapping.

1. Start Workbench.
2. Select **File > New > Example > Native Sample Project**. Click **Next.**
3. Select **The RKM (Remote Kernel Metrics) Monitor Program** and click **Finish**.
4. In the Project Explorer, right-click the **rkm_monitor_linux** folder and select **Build Project**.
5. Right-click the folder again, select **Run As > Open Run Dialog**, then right-click **Process on Target** and choose **New**.
6. In the Main tab, accept the default name or specify a different one.
7. Choose your target connection from the **Connection to use** pull-down menu.
8. Click **Run**. The **rkm_monitor** starts on the target, as shown in your **Debug** view.

Running the RKM Monitor from the Command Line

You can also start the RKM monitor by specifying it on the command line. To see the various options available, enter the following from the directory containing the **rkm_monitor_linux** executable that you built:

```
$ ./rkm_monitor_linux -help
```

For example, to start the monitor with an index value of 125 instead the default 127:

```
$ ./rkm_monitor_linux -index=125 &
```

In this way you can run multiple monitors, for example, configuring one to collect a few signals for all processes at a low sampling frequency, and another to sample a complete set of metrics for a few processes at a high frequency.

By selecting the signals to monitor, you can reduce the memory, CPU, and network resources required to monitor the large default set of signals. Also, you can modify the included **rkm_monitor_linux** source to monitor specific signals not available by the default configuration, or even monitor specific portions of an application.

As another example, you might just want to monitor memory usage for the root user, taking 10 samples every second:

```
$ ./rkm_monitor_linux -samples=10 -processes user=root -sysmetrics memory &
```

When you attach Data Monitor to the RKM monitor invoked as shown, you will only be able to view root memory usage.

Attaching Data Monitor to the RKM Monitor

To view remote kernel metrics in the Workbench Data Monitor GUI:

1. With **rkm_monitor** running on the target, start **Data Monitor** in Workbench by clicking the **Launch Data Monitor** icon (in the main toolbar, approximately between **Search** and **Project** in the menubar).
2. Select the correct target connection from the pull-down menu.
3. Enter 127 as the index value.

   This is the default for Data Monitor with the RKM monitor. Use a different index number if you want to run another monitor on the target; this allows you to run multiple monitors on the target, selecting them by index number.
4. Click **OK**.

   A Data Monitor window opens.

Using Data Monitor to View Remote Kernel Metrics

Your main navigation tool for Data Monitor is the Signals Tree in the upper-left corner. As an example of how to use Data Monitor with remote kernel metrics, use the following procedure.

1. Click the **Connect to Target** icon (the first in the toolbar) and select your target if you are not already connected to it.
2. Expand **Live >** *targetname* **> System > time**.
3. Check the **time** group to automatically check all of the metrics under **time** (**user**, **lowuser**, **system**, **idle**) which then display in units of jiffies (1/100th of a second CPU time) spent executing user, low-priority user, and system tasks, as well as the number of jiffies spent idle.
4. Hover your mouse over the Data Monitor toolbar to find the **Zoom to Fit** icon (near the right-most icon). Click it to contain all the signals in the graph window. Note that each signal has a unique color associated with it and this color is used for the lines in the graph. The MiniMonitor view lists the current values of monitored signals; the MiniDump view lists the value of each signal at each sampling.

By default, when any new processes are started, they are added to the monitor. Note, however, that the buffer is reset when new signals are added, so you lose the history of what you had been monitoring. You can avoid this by, for example, monitoring only your own processes with appropriate command line options as described in *Running the RKM Monitor from the Command Line*, p.307.

For more information on the use of Data Monitor, see the *Wind River Workbench Data Monitor User's Guide*.

18.7 Run/Debug Preferences

For more information, see *Wind River Workbench User Reference: Debug View.*

19 Analyzing Core Files

19.1 Introduction 309
19.2 Acquiring Core Dump Files 310
19.3 Attaching Workbench to a Core File 311

19.1 Introduction

You can configure your target system to save status and memory information on programs that crash for specific reasons. For example, you can specify that the information should be saved if a process exceeds some size, or tries to access memory outside of its allowed memory space. This information is then saved on the target in a file called the core file, or core dump, typically named **core.***pid* where *pid* is the process ID of the program that crashed. The core dump is an ELF file that contains an image of the process memory space, details of the process state, and additional information at the time of the crash. You can then transfer the core file to your host and analyze the cause of the crash using the Workbench debugger at any time.

19.2 Acquiring Core Dump Files

With the bash shell, you control the creation of core dumps with the **ulimit** command. For example, to determine your current settings enter:

```
target_$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) 20
pending signals                 (-i) 1024
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
max user processes              (-u) 11768
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
target_$
```

Most Linux systems come with the default setting to not allow core file generation. In the example this is shown as a core file size setting of **0**. You can turn on core file generation by specifying a maximum size or **unlimited** as the core file size, for example:

```
target_$ ulimit -c unlimited
```

Certain conditions, for example when a program tries to access restricted memory, will then generate core dumps, as in:

```
target_$ ./segtest.out
Segmentation fault (core dumped)
```

You can also turn on core file generation by setting a core file size (**unlimited** or as specified), and then cause a core file to be generated by setting a limit on some condition. For example:

```
target_$ ulimit -c
0
target_$ ulimit -c 12000
target_$ ulimit -c
12000
target_$ ulimit -m
unlimited
target_$ ulimit -m 2000
target_$ ulimit -m
2000
```

The shell is now set to generate a core file of a maximum size of 12000 KB if the memory size for a process exceeds 2000 KB.

NOTE: Your **ulimit** settings entered at the command line apply to the current shell only. If you want to continue to generate core dumps across logins, add the **ulimit** commands to a shell startup file, for example to **.bash_login**.

Transfer the core dump to your host if it is not already on a file system that your target mounts from the host. You can now analyze the core file using the Workbench debugging tools.

19.3 Attaching Workbench to a Core File

Use the Remote Systems view to create a connection to the core file. You do not need to be connected to the target if you have access to the target file system because core file analysis takes place on the host. Note that when you are using Workbench to analyze the core file, you are not debugging an active process, you are only examining the state and conditions of a process at the moment it failed to determine the cause of the failure.

1. Click the **Create a New Target Connection** button in the Remote Systems view, select **Wind River Linux Application Core Dump Target Server Connection**, and click **Next**.
2. Enter or browse to the path for the core dump file you wish to analyze in the **Core dump file** field.
3. You can enter the CPU number for **Force CPU number** and the version of Linux for **Force OS version**. The CPU number for your CPU can be found in the text file **$WIND_FOUNDATION_PATH/resource/target/architecturedb**. For example, the **architecturedb** file shows that the CPU number to enter for an XScale CPU is **1120**:

```
...
[CPU_1120]
cpuname = xscale
cpuFamilyName = XSCALE
...
```

The OS version value to enter is **2.6**.

NOTE: The core file does not contain information on the type of CPU or the version of the operating system that it was created with. If you get a **Failed to connect target** error message indicating a target-CPU mismatch, it is likely that specifying the CPU number and OS version will resolve it.

4. Enter or browse to the path of the application image that created the core dump file in the **File** field.
5. The command line your selections have created is displayed at the bottom of the dialog. To add additional options for memory cache size, logging, and symbol loading, click the **Edit** button next to the **Options** field and make your selections. Click **Next**.
6. Specify the location of the target root by clicking **Add** and entering the **Target path** (for example **/**) and the **Host path** (for example **/target/rootfs**), click **OK,** and then click **Next**.
7. Click **Next** in the **Target Refresh Dialog** box and then click **Finish** in the **Connection Summary** box.

   Your core file connection appears in the Remote Systems view.

You can now connect to the core dump by right-clicking the "stopped" process in the Remote Systems view and selecting **Attach to Process**. The Debug view will show the debugger attached to the process at the point of the failure and an editor window will open at the error location in the source file.

Core File Analysis

You can now perform various activities on the core file, for example view a stack trace, a memory dump, a thread list, local and global variables, and register values. But remember this is only a read-only view of the process at the time of the core dump.

Ending the Session

To end the core file debugging session, disconnect in the Debug view and disconnect in the Remote Systems view.

PART VII

Updating

20 Integrating Plug-ins 315

21 Using Workbench in an Eclipse Environment . 323

22 Using Workbench with Version Control 329

20 Integrating Plug-ins

20.1 Introduction 315

20.2 Finding New Plug-ins 316

20.3 Incorporating New Plug-ins into Workbench 316

20.4 Disabling Plug-in Functionality 320

20.5 Managing Multiple Plug-in Configurations 320

20.6 Installing JDT for Third-Party Plug-ins and Debugging 321

20.1 Introduction

Since Wind River Workbench is based on Eclipse, you can incorporate new modules into Workbench without having to recompile or reinstall it. These new modules are called *plug-ins*, and they can deliver new functionality and tools to your copy of Wind River Workbench.

Many developers enjoy creating new plug-ins and sharing their creations with other Eclipse users, so you will find many Web sites with interesting tools and programs available for you to download and incorporate into your Workbench installation.

For plug-ins dependent on Java Development Tools (JDT), see *20.6 Installing JDT for Third-Party Plug-ins and Debugging*, p.321, for help downloading the JDT.

20.2 Finding New Plug-ins

In addition to the Eclipse Web site, **http://www.eclipse.org**, many other Web sites offer a wide variety of Eclipse plug-ins. Here are a few:

- **http://www.eclipse-plugins.info/eclipse/plugins.jsp**
- **http://www.eclipseplugincentral.com/**
- **http://eclipse-plugins.2y.net/eclipse/**
- **http://www.sourceforge.net/**

20.3 Incorporating New Plug-ins into Workbench

Many developers who download plug-ins prefer to create a new directory for each one, rather than unzipping the files directly into their Workbench installation directory. There are many advantages to this approach:

- The default Workbench installation does not change.
- You do not lose any of your plug-ins if you update or reinstall Workbench.
- Plug-ins do not overwrite each other's files.
- You know which files to replace when an update to the plug-in is available.

20.3.1 Creating a Plug-in Directory Structure

To make your plug-ins easier to manage, create a directory structure for them outside your Workbench installation directory.

1. Create a directory to hold your plug-ins. It can have any descriptive name you want, for example, **eclipseplugins**.
2. Inside this directory, create a directory for each plug-in you want to install. These directories can also have any descriptive name you want, for example, **clearcase**.

NOTE: Before continuing, download the plug-in's **.zip** or other archive file and look at its contents. Some plug-ins provide the **eclipse** directory structure and the **.eclipseextension** file for you, others do not.

- If the destination path for the files begins with **eclipse**, and you see an **.eclipseextension** file in the list, you may skip the rest of this section and extract the plug-in's files into the directory you created in step 2.
- If the destination path begins with **plugins** and **features**, then you must complete the rest of the steps in this section.

3. Inside each plug-in directory, create a directory named **eclipse**. This directory *must* be named eclipse, and a separate **eclipse** directory is required inside each plug-in directory.
4. Inside each eclipse directory, create an empty file named **.eclipseextension**.

   This file *must* be named **.eclipseextension** (with no **.txt** or any other file extension). A separate **.eclipseextension** file is required inside each **eclipse** directory.
5. Extract your plug-in into the **eclipse** directory. Two directories, called **features** and **plugins**, appear in the directory alongside the **.eclipseextension** file.

NOTE: For any plug-in to work properly, its **features** and **plugins** directories as well as an empty file called **.eclipseextension** *must* be located inside a directory called **eclipse**.

20.3.2 Installing a ClearCase Plug-in

Once you have created a plug-in directory structure and have found a plug-in you want to use with Workbench, download and install it according to the instructions provided by the plug-in's developer (almost every plug-in comes with release notes containing installation instructions).

This section will show you how to download and install a plug-in on Windows.

Downloading the IBM Rational ClearCase Plug-in

Wind River recommends the IBM Rational ClearCase plug-in. To install it:

1. Follow steps 1 and 2 in *20.3.1 Creating a Plug-in Directory Structure*, p.316.

   For the purposes of this example, name the top-level directory **eclipseplugins**, and name the plug-in directory **clearcaseIBM**.

   The IBM ClearCase plug-in creates the **eclipse** directory and the **.eclipseextension** file for you.

2. Navigate to **http://www-128.ibm.com/developerworks/rational/library/content/03July/2500/2834/ClearCase/clearcase_plugins.html**. This link was updated 31 January 2007.

NOTE: The IBM links change from time to time. Currently, the above link works, as does the following alternate: **http://www.ibm.com/developerworks/rational/downloads/07/cc_eclipse3_2/clearcase_plugins.html**. This link was updated 02 January 2007.

3. Click the **Download** or **HTTP** link to the right of the appropriate download description of the package, depending on whether you are using Linux or Windows:

   - Adapter V 7.0.0.x for Eclipse 3.2: Linux
   - Adapter V 7.0.0.x for Eclipse 3.2: Windows

   Notice that the corresponding titles in the alternate link are as follows, but they result in downloading the same Linux or Windows file:

   - IBM Rational ClearCase SCM Adapter V 7.0.0.x for Eclipse 3.2: Linux
   - IBM Rational ClearCase SCM Adapter V 7.0.0.x for Eclipse 3.2: Windows

4. Extract the **.zip** file to your **/eclipseplugins/clearcaseIBM** directory.

   The **eclipse** directory is created for you, and inside are two directories, called **features** and **plugins**, alongside the **.eclipseextension** file.

Adding Plug-in Functionality to Workbench

1. Before starting Workbench, make sure that the ClearCase tools directory is in your path:
   - **/usr/atria/bin** for Linux or Solaris
   - **C:\atria\ClearCase\bin** for Windows (this may vary)
2. Start Workbench, then select **Help > Software Updates > Manage Configuration**.

   The **Product Configuration** dialog appears.
3. Select **Add an Extension Location** in the Wind River Workbench pane.
4. Navigate to your **eclipseplugins/***plug-in***/eclipse** directory. Click **OK**.
5. Workbench will ask if you want to restart. To properly incorporate ClearCase functionality, click **Yes**.

Incorporating the IBM Rational Plug-in

When Workbench restarts, activate the plug-in:

1. Select **Window > Customize Perspective**.
2. In the **Customize Perspective** dialog, switch to the **Commands** tab.
3. Select the **ClearCase** option in the **Available command groups** column, then click **OK**.

   A new **ClearCase** menu and icons appear on the main Workbench toolbar.
4. From the **ClearCase** menu, select **Connect to Rational ClearCase** to activate ClearCase functionality.

After you have activated the plug-in:

- Select **Window > Preferences > Team > ClearCase SCM Adapter** to configure it.
- See **Help > Help Contents > Rational ClearCase SCM Adapter** for more information about using it.

For more information about ClearCase functionality, refer to your ClearCase product documentation.

20.4 Disabling Plug-in Functionality

You can disable plug-in functionality without uninstalling the downloaded files. This gives you the opportunity to re-enable them at a later time if you want.

1. To disable a plug-in, select **Help > Software Updates > Manage Configuration**.

   The **Product Configuration** dialog appears.

2. In the left column, open the folder of the plug-in you want to uninstall, select the plug-in itself, then click **Disable**.

3. Workbench will ask if you want to restart. To properly disable the plug-in's functionality, click **Yes**.

20.5 Managing Multiple Plug-in Configurations

If you have many plug-ins installed, you may find it useful to create different configurations that include or exclude specific plug-ins.

When you make a plug-in available to Workbench, its extension location is stored in the Eclipse configuration area.

When starting Workbench, you can specify which configuration you want to start by using the **-configuration** *path* option, where *path* represents your Eclipse configuration directory.

On Windows:

From a shell, type:

```
% cd installdir\workbench-3.x\wrwb\platform\eclipse\x86-win32\bin
% .\wrwb.exe -configuration path
```

On Linux and Solaris:

Use the option as a parameter to the **startWorkbench.sh** script:

```
% ./startWorkbench.sh -configuration path &
```

For more information about using **-configuration** and other Eclipse startup parameters, see **Help > Help Contents > Wind River Partners Documentation > Eclipse Workbench User Guide > Tasks > Running Eclipse**.

20.6 Installing JDT for Third-Party Plug-ins and Debugging

Previous versions of Workbench included a modified version of the Java Development Tools (JDT), but this has been removed for better integration with Eclipse and certain third party packages.

However, some third party plug-ins have a dependency on JDT.

To use the Workbench Update Manager to install the JDT:

1. Open the Update Manager by selecting **Help > Software Updates > Find and Install**.
2. Select **Search for new features to install**, then click **Next**.
3. Select **The Eclipse Project Updates**, check **Automatically select mirrors**, and click **Finish**.

   The Search Results dialog opens.
4. Check **Show the latest version of a feature only**.
5. Expand **The Eclipse Project Updates**, then **Eclipse SDK 3.3.1**.
6. Check **Eclipse Java Development Tools 3.3.1.r331** and click **Next**.
7. Accept the JDT license agreement, then click **Next**.
8. Click **Finish**.

   The Feature Verification dialog eventually opens.
9. Click **Install All**.
10. Click **Yes** to restart Workbench.

You can also use the Update Manager to search for new features and other updates.

21

Using Workbench in an Eclipse Environment

21.1 Introduction 323

21.2 Recommended Software Versions and Limitations 324

21.3 Setting Up Workbench 324

21.4 Using CDT and Workbench in an Eclipse Environment 325

21.1 Introduction

It is possible to install Workbench in a standard Eclipse environment, though some fixes and improvements that Wind River has made to Workbench will not be available.

21.2 Recommended Software Versions and Limitations

Java Runtime Version

Wind River tests, supports, and recommends using the JRE 1.5.0_08 for Workbench plug-ins.

Wind River adds a package to that JRE version, and not having that package will make the Terminal view inoperable.

Eclipse Version

Workbench 3.0 is based on Eclipse 3.2. Wind River patches Eclipse to fix some Eclipse debugger bugs. These fixes will be lost when using a standard Eclipse environment.

See the getting started for your platform for supported and recommended host requirements for Workbench 3.0.

Defaults and Branding

Eclipse uses different default preferences from those set by Workbench. The dialog described in *21.3 Setting Up Workbench*, p.324 allows you to select whether to use Workbench preferences or existing Eclipse preferences.

In a standard Eclipse environment, the Eclipse branding (splash screen, welcome screen, and so on) is used instead of the Wind River branding.

21.3 Setting Up Workbench

This setup requires a complete Eclipse and Workbench installation. Follow the respective installation instructions for each product.

1. From within Workbench, select **Help > Register into Eclipse**. The Register into Eclipse dialog appears.
2. In the **Directory** field, type in or **Browse** to your Eclipse 3.2 directory.

3. In the **Registration Options** section, select **Use Wind River default preferences**, or leave it cleared to maintain existing Eclipse preferences.

   If you decide to use Wind River default preferences, some changes you will notice are that autobuild is disabled, and the Workbench Application Development perspective and help home become the defaults.

4. If you decided to maintain existing Eclipse preferences you can still use the much faster Wind River (index based) search engine by leaving **Use Wind River search engine** selected. To use the Eclipse default search engine, unselect it.

5. If you want to track the installation process, leave **Log installation process** selected (click **Browse** to change the path where the file should be created). Uncheck it if you do not want Workbench to create a log file.

6. When you are done, click **Finish**. Workbench will be available the next time you launch Eclipse. No special steps are necessary to launch Eclipse.

NOTE: Any errors discovered during installation appear in the Error Log view.

21.4 Using CDT and Workbench in an Eclipse Environment

The following tips will help you understand how to use Eclipse C/C++ Development Tooling (CDT) and Workbench together in the same Eclipse environment.

NOTE: When starting Eclipse after registering Workbench, you will see three errors in the Error Log.

These errors are not a problem. They appear because Workbench ships some CDT plug-ins that are already on your system, and Eclipse is reporting that the new ones will not be installed over the existing ones.

21.4.1 Workflow in the Project Explorer

Some menus and actions are slightly different when using CDT and Workbench together.

Application Development Perspective (Workbench)

CDT projects appear in this perspective along with Workbench projects.

Building CDT Projects

The context menu of the Project Explorer contains entries for **Build Project** and **Rebuild Project**, but the **Rebuild Project** entry executes a normal build for CDT projects. The **Clean Project** entry is missing for CDT projects.

Running Native Applications

The **Run Native Application** menu is enabled for CDT projects. When executed, it creates a Workbench Native Application launch with correct parameters. Because Workbench Native Application launches do not support debugging, to debug your application you must create a CDT **Local C/C++ Application** launch from the **Run > Run As** menu.

Selecting Projects to Build

When selecting multiple projects (including Workbench and CDT projects) and executing any build action, the build action is only executed on Workbench projects.

Displaying File and Editor Associations

The Workbench Project Explorer displays icons for the default editor of a file, if file associations have been defined. If CDT is the default editor, the corresponding icons will also show up in the Application Development perspective.

C/C++ Perspective (CDT)

Source Analysis

Source analysis is available from the context menu of the Project Explorer.

Building Workbench Projects

CDT **Build Project** and **Clean Project** actions are enabled for Workbench projects, and they execute the appropriate build commands correctly.

Working with Workbench Binary Targets

There are no actions to directly run, debug or download a Workbench project's binary target in this perspective.

21.4.2 Workflow in the Build Console

Application Development Perspective (Workbench)

When adding a CDT project as a sub-project (project reference) to a Workbench project, the **Clear Build Console** flag is ignored when executing a build on this project.

C/C++ Perspective (CDT)

Executing a build on a Workbench project from this perspective correctly opens the Workbench Build Console.

General

When navigating to errors from the Workbench Build Console or the Problems view, the file containing the error opens in the assigned editor.

21.4.3 Workflow in the Editor

Opening Files in an Editor

The editor that should be used for files cannot be determined. It depends on the settings defined in the appropriate **plugin.xml** files, and on the order in which the Workbench and CDT plug-ins are loaded.

Only one default editor can be associated with each file type, and it is the same for both perspectives. Files can be opened with the **Open With** menu, allowing you to select the editor. When executed, that editor is associated with, and becomes the default for, this specific file.

NOTE: To assign a default editor for all files with a given signature, you must define a file association in the preferences by selecting **Window > Preferences**, then choosing **General > Editors > File Associations**.

For example, to add a default editor for all ***.c** files, click **Add** and enter ***.c**. The list of available editors appears. Select one, then click **Default**.

21.4.4 Workflow for Debugging

Workbench and CDT Perspectives

Regardless of any direct file association created using the **Open With** command, the default editor opens when debugging a file.

For example, associating ***.c** files with the default Workbench editor opens the Workbench editor in the CDT Debug and the Workbench Device Debug perspectives.

The reverse is also true: if you associate a file type with the CDT editor, it will open when those files are debugged even if you have made an association with a different editor using **Open With**.

22

Using Workbench with Version Control

22.1 Introduction 329

22.2 Using Workbench with ClearCase Views 329

22.1 Introduction

This chapter provides tips on using Workbench with version-controlled files, which Workbench project files you should add to version control when archiving your projects, and how to manage build output when your sources are version controlled.

22.2 Using Workbench with ClearCase Views

When using Workbench with ClearCase dynamic views, create your workspace on your local file system for best performance. For recommendations about setting up your workspaces and views, see **Help > Help Contents > Rational ClearCase SCM Adapter > Concepts > Managing workspaces**.

Wind River does not recommend that you place the Eclipse workspace directory in a view-private directory. If you create projects in the default location under the

workspace directory, ClearCase prompts you to add the project to source control. This process requires all parent directories to be under source control, including the workspace directory.

Instead, create workspace directories outside of a ClearCase view. If you want to create projects under source control, you should clear the **Create project in workspace** check box in the project creation dialog and then navigate to a path in a VOB.

In addition, you should also redirect all build output files to the local file system by changing the **Redirection root directory** in the **Build Properties > Build Paths** tab of your product. All build output files such as object files and generated Makefiles will be redirected.

For more information about the redirecting build output and the redirection root directory, open the build properties dialog, press the help key for your host, and see the *Build Paths* section.

22.2.1 Adding Workbench Project Files to Version Control

To add Workbench project files to version control without putting your workspace into a ClearCase view, check-in the following automatically generated files along with your source files:

Project File	Description
.project	Eclipse platform project file containing general information about the project.
.wrproject	Workbench project file containing mostly general build properties.
.wrfolder	Workbench project file containing folder-level build properties (located in subfolders of your projects).
.wrmakefile	Workbench managed build makefile template used to generate Makefiles.
***.makefile**	Workbench managed build extension makefile fragments (e.g some Platform projects)

For user-defined projects, all Makefile files need to be version controlled, too.

You should avoid manually adding source files to any build macro in any project type containing absolute paths—they should be substituted by environment variables (provided by **wrenv** for example) wherever possible.

NOTE: The **.metadata** directory should not be version controlled, as it contains mostly user- and workspace-specific information with absolute paths in it.

For more information about IBM Rational ClearCase, see **http://www-130.ibm.com/developerworks/rational/products/clearcase**.

22.2.2 Choosing Not to Add Build Output Files to ClearCase

After installing the ClearCase plugin, you may be prompted to add any build output files to ClearCase.

There are two ways to avoid this if you wish:

1. Using Workbench Preferences.
   a. Open the **Window > Preferences > Team > ClearCase SCM Adapter** preferences page.
   b. From the **When new resources are added** pull-down list, select **Do nothing**.
2. Using **Derived Resource** option.
   a. Configure your build so the build output goes into one (or a few) well-known directories such as **bin** or **output**.
   b. Check in the empty **bin** or **output** directories into ClearCase.
   c. In the Project Explorer, right-click the directory you checked in, select **Properties**, and on the **Info** page, select **Derived**.
   d. From now on, the Clearcase plug-in will not prompt you about **Derived** resources.

NOTE: If you use Workbench managed builds, they will automatically mark the build output directories as derived so ClearCase will not try to add the build output files to source control. If you use a different builder, you may have to configure it to mark resources as derived.

PART VIII

Reference

A Troubleshooting 335

B Workbench and Command-Line Techniques ... 365

C Installing on Network Servers 371

D Configuring a Wind River Proxy Host 375

E Command-line Updating of Workspaces 383

F Command-line Importing of Projects 387

G Starting usermode-agent at Boot Time 391

H Glossary 393

A Troubleshooting

A.1 Introduction 335

A.2 Startup Problems 336

A.3 General Problems 339

A.4 Fixing Indexer Issues 341

A.5 Hints about Performance and Breakpoints 344

A.6 Error Messages 346

A.7 Error Log View 355

A.8 Error Logs Generated by Workbench 355

A.9 Technical Support 363

A.1 Introduction

This appendix displays some of the errors or problems that may occur at different points in the development process, and what steps you can take to correct them. It also provides information about the log files that Workbench can collect, and how you can create a ZIP file of those logs to send to Wind River support.

If you are experiencing a problem with Workbench that is not covered in this chapter, please see the *Wind River Workbench Release Notes* for your platform.

A.2 Startup Problems

This section discusses some of the problems that might cause Workbench to have trouble starting.

Workspace Metadata Is Corrupted

If Workbench crashes, some of your settings could get corrupted, preventing Workbench from restarting properly.

1. To test if your workspace is the source of the problem, start Workbench, specifying a different workspace name.

   On Windows

   Select **Start > Programs > Wind River > Workbench 3.0 > Wind River Workbench 3.0**, then when Workbench asks you to choose a workspace, enter a new name (**workspace2** or whatever you prefer).

   Or, if the Workbench startup process does not get all the way to the Workspace Launcher dialog box, or does not start at all, start it from a terminal window:

   > *installDir*`\workbench-3.0\wrwb\platform\eclipse\x86-win32\bin\wrwb.exe -data` *newWorkspace*

   On Linux or Solaris

   Start Workbench from a terminal window, specifying a new workspace name:

   > `./startWorkbench.sh -data` *newWorkspace*

   NOTE: For more information on Workbench startup options, see **Help > Help Contents > Wind River Partners Documentation > Eclipse Workbench User Guide > Tasks > Running Eclipse**.

2. If Workbench starts successfully with a new workspace, exit Workbench, then delete the **.metadata** directory in your original Workbench installation (*installDir***/workspace/.metadata**).
3. Restart Workbench using your original workspace. The **.metadata** directory will be recreated and should work correctly.
4. Because the **.metadata** directory contains project information, that information will be lost when you delete the directory.

To re-create your project settings, reimport your projects into Workbench (**File > Import > Existing Project into Workspace**).

.workbench-3.0 Directory Is Corrupted

1. To test if your *homeDir*/**.workbench-3.0** directory is the source of the problem, rename it to a different name, then restart Workbench.

NOTE: Make sure you rename the *homeDir*/**.workbench-3.0** directory, not the *installDir*/**workbench-3.0** directory.

2. If Workbench starts successfully, exit Workbench, then delete the old version of your *homeDir*/**.workbench-3.0** directory (the one you renamed).
3. Restart Workbench. The *homeDir*/**.workbench-3.0** will be recreated and should work correctly.
4. Because the **.workbench-3.0** directory contains Eclipse configuration information, any information about manually configured Eclipse extensions or plug-ins will be lost when you delete the directory.

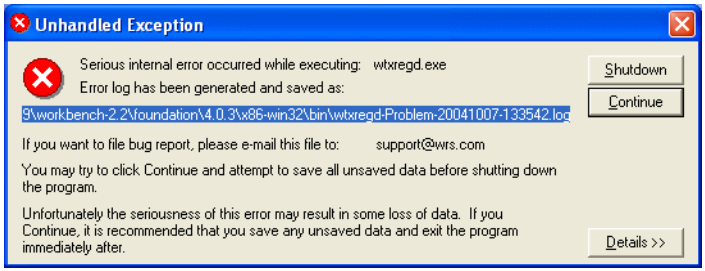
   To make them available again within Workbench, re-register them (**Help > Software Updates > Manage Configuration**).

Registry Unreachable (Windows)

When Workbench starts and it does not detect a default Wind River registry, it launches one. After you quit Workbench, the registry is kept running since it is needed by all Wind River tools. You do not need to ever kill the registry.

If you do stop it, however, it stores its internal database in the file *installDir*/**workbench-3.0/foundation/.wind/wtxregd.***hostname*.

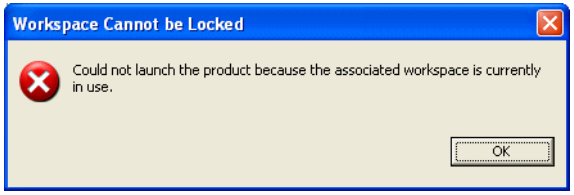
If this file later becomes unwritable, the registry cannot start, and Workbench will display an error.


Unhandled Exception
Serious internal error occurred while executing: wtxregd.exe
Error log has been generated and saved as:
9\workbench-2.2\foundation\4.0.3\x86-win32\bin\wtxregd-Problem-20041007-133542.log
If you want to file bug report, please e-mail this file to: support@wrs.com
You may try to click Continue and attempt to save all unsaved data before shutting down the program.
Unfortunately the seriousness of this error may result in some loss of data. If you Continue, it is recommended that you save any unsaved data and exit the program immediately after.
Shutdown
Continue
Details >>

This error may also occur if you install Workbench to a directory to which you do not have write access, such as installing Workbench as an administrator and then trying to run it as yourself.

Workspace Cannot Be Locked (Linux and Solaris)

If you start Workbench and select a workspace, you may see a **Workspace Cannot be Locked** error.



There are three possible causes for this error:

1. Another user has opened the same workspace. A workspace can only be used by one user at a time.
2. You installed Workbench on a file system that does not support locking.

   Use the following command at a terminal prompt to start Workbench so that it creates your workspace on a file system which does allow locking, such as a directory on a local disk:

   `./startWorkbench.sh -configuration` *directory that allows locking*

   For example:

   `./startWorkbench.sh -configuration /usr/local/`*yourName*

NOTE: For more information on Workbench startup options, see **Help > Help Contents > Wind River Partners Documentation > Eclipse Workbench User Guide > Tasks > Running Eclipse**.

3. On some window managers (e.g. gnome) you can close the window without closing the program itself and deleting all running processes. This results in running processes maintaining a lock on special files in the workspace that mark a workspace as open.

   To solve the problem, kill all Workbench and Java processes that have open file handles in your workspace directory.

Pango Error on Linux

If the file **pango.modules** is not world readable for some reason, Workbench will not start and you may see an error in a terminal window similar to

```
** (<unknown>:21465): WARNING **: No builtin or dynamically loaded modules
were found. Pango will not work correctly. This probably means there was an
error in the creation of:
   '/etc/pango/pango.modules'
You may be able to recreate this file by running pango-querymodules.
```

Changing the file's permissions to **644** will cause Workbench to launch properly.

A.3 General Problems

If you are experiencing a problem with Workbench that is not covered in this chapter, please see the *Wind River Workbench Release Notes* for your platform.

A.3.1 Help System Does Not Display on Linux

Workbench comes pre-configured to use Mozilla on Linux, and it expects it to be in your path. If Mozilla is not installed or is not in your path, you must install and set the correct path to the browser or Workbench will not display help or other documentation.

To manually set the browser path in Workbench:

1. Select **Window > Preferences > Help**.
2. Click **Custom Browser (user defined program)**, then in the **Custom Browser command field** type or browse to your browser launch program, click **OK**.

   Sample browser launch commands are **"/usr/bin/firefox" %1**, **"kfmclient openURL %1"**, and **"/opt/mozilla/mozilla %1"**. Enter the command line as appropriate for your browser.

A.3.2 Help System Does Not Display on Windows

The help system can sometimes fail to display help or other documentation due to a problem in McAfee VirusScan 8.0.0i (and possibly other virus scanners as well).

For McAfee VirusScan 8.0.0i, the problem is known to be resolved with patch10 which can be obtained from Network Associates. As a workaround, the problem can be avoided by making sure that McAfee on-access-scan is turned **on** and allowed to scan the **TEMP** directory as well as ***.jar** files.

More details regarding this issue have been collected by Eclipse Bugzilla #87371 at **https://bugs.eclipse.org/bugs/show_bug.cgi?id=87371**.

A.3.3 Resetting Workbench to Its Default Settings

If Workbench crashes, some of your settings could get corrupted, preventing Workbench from restarting properly. To reset all your settings to their defaults, delete your *$HOME*/**.workbench-3.0** directory which will be recreated when Workbench restarts.

CAUTION: Remove the directory **.workbench-3.0** (begins with a "dot") in your home directory, not the directory **workbench-3.0** in the Workbench installation directory.

A.4 Fixing Indexer Issues

If you encounter troubles with source navigation tools for certain files, resulting in missing or incorrect information in Include Browser, Type Hierarchy, Call Tree, code-completion, Outline view, and so on, these files have very likely not been parsed, or parsed with wrong include paths or symbols.

This section describes various solutions to indexer and source analysis issues.

A.4.1 Indexing Problems with Managed Projects

If the files are not part of the current build-target, or all build targets in the project are empty:

- Add the files to the build-target. This changes the build, so you need to update the index afterwards.
- Activate the **Index All** files option on the **C/C++ General > Indexer** project property page, so that all files of the project are parsed).

A.4.2 Indexing Problems with User-defined Projects

The following problems might show up specifically in user-defined projects.

Source Files Not Yet Built

If the source files have not yet been built:

- Build all source files of your project and then update the index.
- Disable build-output analysis on the **Discovery** tab page of the **C/C++ General > Paths and Symbols** project property page, then click **Clear**. This disables build-driven indexer setup and causes all files of the project to be parsed, except standalone header files. You need to update the index afterwards.
- Activate the **Index All files** option on the **C/C++ General > Indexer** project property page. This disables build-driven indexer setup and causes all files of the project to be parsed.

Unsuccessful Built Output Analysis

If build-output analysis has not been successful (for example, because of unsupported make rules):

- Make sure that the `--no-print-directory` make option is not set.
- Disable build-output analysis on the **Discovery** tab page of the **C/C++ General > Paths and Symbols** project property page, and click **Clear**. This disables build-driven indexer setup and causes all files of the project to be parsed (except standalone header files). You need to update the index afterwards.
- Activate the **Index All files** option on the **C/C++ General > Indexer** project property page. This disables build-driven indexer setup and causes all files of the project to be parsed. Note that this increases the memory consumption of Workbench.

A.4.3 Other Indexing Problems

The problems in this section may appear regardless of whether the project is managed or user-defined.

Files Excluded in the GUI

If the files have been excluded via the **Sources / Filters** tab page on the **C/C++ General > Paths and Symbols** project property page:

1. Open the **Sources / Filters** tab page on the **C/C++ General > Paths and Symbols** project property page.
2. Expand any source folder and check the filters below.
3. Click **Edit filter** data to change exclusion filters.

Note that filtered paths are not parsed, and that you need to upate the index after making these changes.

Outdated Index

If the index is not up to date:

- Right-click the project in the Project Explorer and select **Index > Update with Modified Files** in order to parse modified and newly added files.
- Right-click the project in the Project Explorer and select **Index > Rebuild** in order to clear the index and re-parse all files in the current build scope (for build-driven setup) or all files in the project (no build target, or the **Index all files** option is enabled for a project, or build output analysis is disabled or did not return any results).

NOTE: Changing include paths or symbols, for example, via the **Build Properties** page or the **Paths and Symbols** property page, only has an immediate affect on parsing modified and newly added files. Rebuild the index manually if you need accurate source navigation.

Incorrect Include Paths and Symbols

If the include paths and symbols are not set correctly:

- For managed projects, build properties are passed to the indexer. If the project successfully builds, the include paths and symbols are most likely correct. Let Workbench detect include paths to resolve include directives (using the ISP wizard, see earlier in this section), or specify additional paths manually in the build properties.
- For user-defined projects, the indexer is set up automatically with include paths found through inspection of source-code right after project creation. To get better results, you can perform a full build with enabled build-output analysis (for that purpose, right-click the project in the Project Explorer, then select **C/C++ General > Paths and Symbols**, and switch to the **Discovery** tab page).

Troubles Parsing Source Code

If the indexer has troubles parsing the source-code:

- Open **Window > Preferences** and go to section **C/C++ > Language Mappings**. Change the language mappings for **C Header File** and **C Source File** to Wind River Compiler C++ Dialect.

If the problem still appears, send a report to Wind River, including log files (in the main menu, select **Help > Collect Log Files**). If possible, include a small project that shows the problem.

A.5 Hints about Performance and Breakpoints

Wind River Linux platform projects are very large, and can present a challenge to Workbench performance.

If you follow the standard method of using the pre-build packages and even the kernel, this may not be a problem. If you perform a **make fs**, or build several kernels, then these steps can create up to 28,000 files each project for Workbench to manage and track.

Module Optimization Levels and Jumping Program Counters

The kernel will be built with various optimization flags set. If you step through kernel code, you will see the program counter marker jump around as you step through the optimized and re-ordered instructions. This is normal, and proves that optimization was applied.

This will also occur for kernel modules if you compile without overriding the optimization level, because by default all kernel modules built against a kernel will inherit its optimization. This can be overcome by building your kernel modules without optimization.

Right-click on the Kernel Module Project, and select **Properties > Build Properties**

Change the Build command to:

```
make COPTIMIZE="-O0"  (that is a dash, a capital letter "O", and a zero)
```

When you next build the module, it will have no (or to be exact, minimum) optimization.

Module Optimization Levels and Skipped Breakpoints

Sometimes you may find behavior like a step-next that does not break on the next step. In this example, the compiler when optimizing the code may in fact create two in-line copies, where the break point gets set in the first copy, but the code may branch into the second copy. This is a limitation of some ELF formats, and a resulting optimization and context awareness issue for the debugger.

You can try switching to mixed assembly and source stepping, where you step the actual assembly instructions, but the best choice is again to remove the optimization for the module and retest.

Manual Refresh of the Build Linked Directory in Workbench

To help the workflow, Workbench has disabled the automatic project refresh after builds so that the user can immediately choose the next task. The consequence is that you need to manually refresh the **build** directory whenever you wish to "see" any new or revised file within, or else the file may not be visible, or the existing file handle in the Project Explorer might be stale.

Previously, platform projects included a link to the project's build directory, allowing the user to browse their local source packages. This however added a huge refresh penalty due to the 10,000 to 40,000 files that can appear in that tree, This link is no longer included by default. If you need to browse the build directory, use Source Analysis or the Quilt patch mechanism, right-click the project and select **Add link to build folder**.

Disabling Workspace Refresh on Startup

You may choose to disable the automatic workspace refresh when starting Workbench, particularly when you have one or more platform projects.

This is done by selecting **Window > Preferences > General > Startup and Shutdown**, and un-checking **Refresh workspace on startup**.

When you need to browse the build tree, you can always manually refresh that directory by right-clicking on the project (or specifically the **build** entry) and selecting **Refresh**.

Workbench Freezes: No Response to Mouse Clicks

Sometimes Workbench will not immediately refresh its screen or respond to mouse or keyboard actions. This is often due to (a) a garbage collection timeout, or (b) running low on real and virtual memory. In Wind River Linux Platform Projects, the heap allocation amount can rise and fall 40Mbytes at a time, and will often rise to 500 MB in size.

You can track this behavior be enabling Workbench heap status monitor, with the command **Windows > Preferences > General > Show Heap Status**. The heap monitor will appear on the lower right-hand corner.

Planting (Inserting) Breakpoints

Sometime it is hard to insert a breakpoint in the middle of a file, and have it actually planted.

One trick is to set a breakpoint on a global routine to bring in the expected source file to the debugger (Workbench may in fact prompt and guide you if you had not properly set up the proper source code mapping). Once that happens, you can then set the breakpoint that you desire. That global routine can be the same routine that contains the desired location, or it could be an entirely unrelated routine, as long as it is in the same C file.

A.6 Error Messages

Some errors display an error dialog box directly on the screen, while others that occurred during background processing only display this icon in the lower right corner of Workbench window.

Hovering your mouse over the icon displays a pop-up with a synopsis of the error. Later, if you closed the error dialog box but want to see the entire error message again, double-click the icon to display the error dialog box or look in the *Eclipse Log*, p.357.

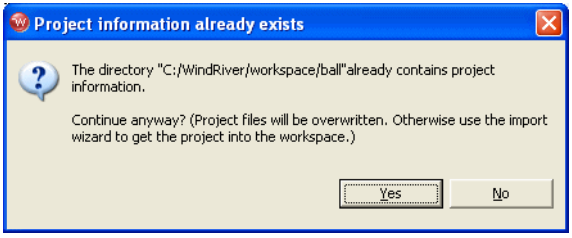
This section explains error messages that appear in each Workbench component.

A.6.1 Project System Errors

For general information about the Project System, see *6. Projects Overview*.

Project Already Exists

If you deleted a project from the Project Explorer but chose not to delete the project contents from your workspace, then you try to create a new project with the same name as the old project, you will see the following error:


Project information already exists
The directory "C:/WindRiver/workspace/ball"already contains project information.
Continue anyway? (Project files will be overwritten. Otherwise use the import wizard to get the project into the workspace.)
Yes
No

If you click **Yes**, your old project contents *will be overwritten* with the new project. If you want to recreate the old project in Workbench, click **No**, then right-click in the Project Explorer, select **Import**, then select **Existing Project into Workspace**.

Type the name of your old project, or browse to the old project directory in your workspace, click **OK**, then click **Finish**. Your old project will appear in the Project Explorer.

Cannot Create Project Files in Read-only Location

When Workbench creates a project, it creates a **.wrproject** file and other metadata files it needs to track settings, preferences, and other project-specific information. So if your source files are in a read-only location, Workbench cannot create your project there.

To work around this problem, you must create a new project in your workspace, then create a folder that links to the location of your source files.

1. Create a User-defined Project in your workspace by selecting **File > New > User-Defined Project**. The Target Operating System dialog box appears.
2. Select a target operating system from the drop-down list, then click **Next**. The Project dialog box appears.
3. Type in a name for your project, select **Create project in workspace**, then click **Next**.
4. Click **Next** to accept the default settings in the next dialog boxes, then click **Finish** to create your project.
5. In the Project Explorer, right-click your new project and select **New > Folder**. The **Folder** dialog box appears.
6. Type in a name for your folder, then click **Advanced** and select the **Link to folder in the file system** check box.
7. Type the path or click **Browse** and navigate to your source root directory, then click **OK** to create the new folder.
8. Click the plus next to the folder to open it, and you will see the source files from your read-only source directory. Eclipse calls items incorporated into projects in this way **linked resources**.

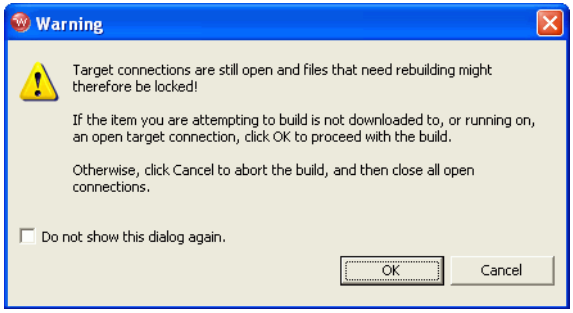
NOTE: This mechanism cannot be used for managed-build projects, only for user-defined projects.

A.6.2 Build System Errors

For general information about the Build System, see *11. Building Projects*.

A.6.3 Building Projects While Connected to a Target

If you right-click a project in the Project Explorer and select **Build Project** while you have a target connection active in the Remote Systems view, you will see this error:


Warning
Target connections are still open and files that need rebuilding might therefore be locked!
If the item you are attempting to build is not downloaded to, or running on, an open target connection, click OK to proceed with the build.
Otherwise, click Cancel to abort the build, and then close all open connections.
Do not show this dialog again.
OK
Cancel

This dialog box warns you that a build may fail because the debugger may still have a lock on your files. You can continue your build by clicking **OK**, but be advised that when you see an error message in the Build Console similar to **dld: Can't create file XXX: Permission denied** you will need to disconnect your target and restart the build.

The best workflow for cases where you continually need to rebuild objects that are in use by your target is as follows:

- Create a launch configuration for your debugging task. When you need to disconnect your target in order to free your images for the build process, the launch configuration allows you to automatically connect, download, and run your process with a single click when the build is finished.

  You can even specify that your project should be rebuilt before it is launched by selecting **Window > Preferences > Run/Debug > Launching**, and then selecting **Build (if necessary) before launching**. For more information about launch configurations, see *17. Launching Programs*.

- When you work with processes, make sure that your process is terminated before you rebuild or relaunch. You can then safely ignore the warning (and check the **Do not show this dialog again** box).

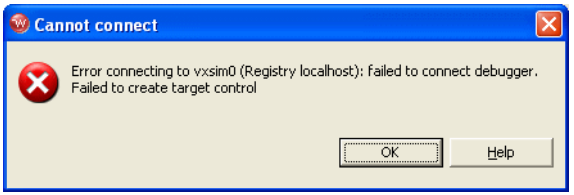
- When you work with Downloadable Kernel Modules or user-built kernel images, just let the build proceed. If the **Link error** message appears, either disconnect your target or unload all modules, then rebuild or relaunch.

A.6.4 Remote Systems View Errors

For information about the Remote Systems view, see *13. Connecting to Targets*.

Troubleshooting Connecting to a Target

If you see the following error:



Or if you have other trouble connecting to your target, try these steps:

1. Check that the target is switched on and the network connection is active. In a terminal window on the host, type:

   `ping` *n.n.n.n*

   where **n.n.n.n** is the IP address of your target.

2. Verify the target **Name/IP address** in the **Edit the Target Connection** dialog box (right-click the target connection in the Remote Systems view then select **Properties**.)

3. Choose the actual target CPU type from the drop-down list if the **CPU type** in the **Edit the Target Connection** dialog box is set to **default from target**.

4. Verify that a target server is running. If it is not:

   a. Open the Error Log view, then find and copy the message containing the command line used to launch the target server.

   b. Paste the target server command line into a terminal window, then press **ENTER**.

   c. Check to see if the target server is now running. If not, check the Error Log view for any error messages.

5. Check if the **dfwserver** is running (on Linux and Solaris, use the **ps** command from a terminal window; on Windows, check the Windows Task Manager). If multiple **dfwservers** are running, kill them all, then try to reconnect.

6. Check that the WDB connection to the target is fully operational by right-clicking a target in the Remote Systems view and selecting **Target Tools > Run WTX Connection Test**. This tool will verify that the communication link is correct. If there are errors, you can use the WTX and WDB logs to better track down what is wrong with the target.

Exception on Attach Errors

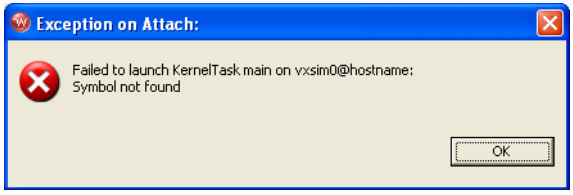
If you try to run a task and the Remote Systems view is unable to comply, it will display an **Exception on Attach** error containing useful information.

Build errors can lead to a problem launching your task or process; if one of the following suggestions does not solve the problem, try launching one of the pre-built example projects delivered with Workbench.

If the host shell was running when you tried to launch your task or process, try closing the host shell and launching again.

Error When Running a Task Without Downloading First

You will see the following error if you try to run a kernel task without first downloading it to your target:



Processes can be run directly from the Project Explorer, but kernel tasks must be downloaded before running. Right-click the output file, select **Download**, fill in the Download dialog box, then click **OK**.

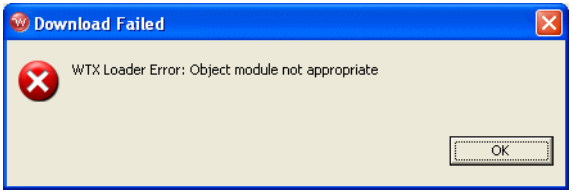
If you see this error and you did download the file, open a host shell for your connection, and try to run the task from the host shell. Type:

```
lkup entrypoint
```

to see if your entry point is there.

Downloading an Output File Built with the Wrong Build Spec

If you built a project with a build spec for one target, then try to download the output file to a different target (for example, you build the project for the simulator, but now you want to run it on a hardware target), you will see this error:

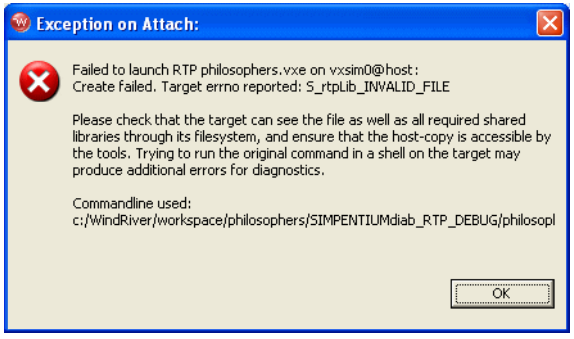


To select the correct build spec, right-click the output file in the Project Explorer, select **Set Active Build Spec**, select the appropriate build spec from the dialog box, then rebuild your project.

Your project should now download properly.

Error if Exec Path on Target Is Incorrect

If the **Exec Path on Target** field of the **Run Real-time Processes** dialog box does not contain the correct target-side path to the executable file (if, for example, it contains the equivalent host-side path instead) you will see this error:


Exception on Attach:
Failed to launch RTP philosophers.vxe on vxsim0@host:
Create failed. Target errno reported: S_rtpLib_INVALID_FILE
Please check that the target can see the file as well as all required shared libraries through its filesystem, and ensure that the host-copy is accessible by the tools. Trying to run the original command in a shell on the target may produce additional errors for diagnostics.
Commandline used:
c:/WindRiver/workspace/philosophers/SIMPENTIUMdiab_RTP_DEBUG/philosopl
OK

If the target-side path looks correct but you still get this error, recheck the path you gave.

Even if you used the **Browse** button to locate the file, it will be located in the host file system. The Object Path Mapping that is defined for your target connection will translate it to a path in the target file system, which is then visible in the Exec Path edit field. If your Object Path Mapping is wrong, the Exec Path will be wrong, so it is important to check.

Troubleshooting Running a Process

If you have trouble running your process from the **Run Process** or **Run Real-time Process** dialog boxes, try these steps:

1. If the error **Cannot create context** appears, verify that the **Exec Path on Target** is a path that is actually visible on the target (and does not contain the equivalent host-side path instead).
   a. Right-click the process executable in the Project Explorer or right-click **Processes** or **Real-time Processes** in the Remote Systems view and select **Run Real-time Process.**
   b. Copy the exec path and paste it into the **Output View > Target Console Tab** (at the bottom of the view). Verify that the program runs directly on the target.
2. If the program runs but symbols are not found, manually load the symbols by right-clicking the process and selecting **Load Symbols**.
3. Check your **Object Path Mappings** to be sure that target paths are mapped to the correct host paths. See *13.5.2 Object Path Mappings*, p.234 for details on setting up your Object Path Mappings.
   a. Open a host shell and type:

      ```
      ls execpath
      ```

      If you have a target shell, type the same command.
   b. In the host shell, type:

      ```
      devs
      ```

      to see if the prefix of the Exec Path (for example, **host:**) is correct.
4. If the Exec Path is correct, try increasing the back-end timeout value of your target server connection (see *Advanced Target Server Options*, p.232 for details).
5. From a target shell or Linux console, try to launch the process.
6. Verify that the kernel node in the Remote Systems view has a small **"s"** added to the icon, indicating that symbols have been loaded for the kernel.

   If not, see that the last line of the **Object Path Mappings** table displays a target path of **<any>** corresponding to a host path of **<leave path unchanged>**.

A.6.5 Launch Configuration Errors

If a launch configuration is having problems, delete it by clicking **Delete** below the **Debug** dialog box **Configurations** list.

If you cannot delete the launch configuration using the **Delete** button, navigate to *installDir*/**workspace/.metadata/.plugins/org.eclipse.debug.core/.launches** and delete the **.launch** file with the exact name of the problematic launch configuration.

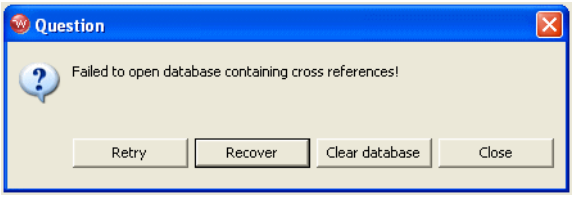
NOTE: Do *not* delete any of the **com.windriver.ide.*.launch** files.

If you click the **Debug** button (or click the **Debug** button from the **Launch Configuration** dialog box) and get a "Cannot create context" error, check the **Exec Path** on the **Main** tab of the **Debug** dialog box to be sure it is correct. Also check your **Object Path Mappings**. See *13.5.2 Object Path Mappings*, p.234 for information about Object Path Mappings.

For general information about launch configurations, see *17. Launching Programs*.

A.6.6 Source Analysis Errors

If at any point Workbench is unable to open the cross reference database, you will see this error:



Reasons the cross reference database may be inaccessible include:

- The database was not closed properly at the end of the last Workbench session running within the same workspace. This happens if the process running Workbench crashed or was killed.
- Various problems with the file system, including wrong permissions, a network drive that is unavailable, or a disk that is full.

You have several choices for how to respond to this error dialog box:

- **Retry** – Perform the same operation again, possibly with the same failure.

- **Recover** – Open the database and attempt a repair operation. This may take some time but you may recover your cross reference data.
- **Clear Database** – Delete the database and create a new one. Cross reference data is lost and your workspace will be reparsed when you open the call tree.
- **Close** – Close the database. No cross reference data is available, nor will it be generated. At the beginning of the next Workbench session, an attempt to open the database will be made again.

A.7 Error Log View

See the *Wind River Workbench User Interface Reference: Error Log View*.

A.8 Error Logs Generated by Workbench

Workbench has the ability to generate a variety of useful log files. Some Workbench logs are always enabled, some can be enabled using options within Workbench, and some must be enabled by adding options to the executable command when you start Workbench.

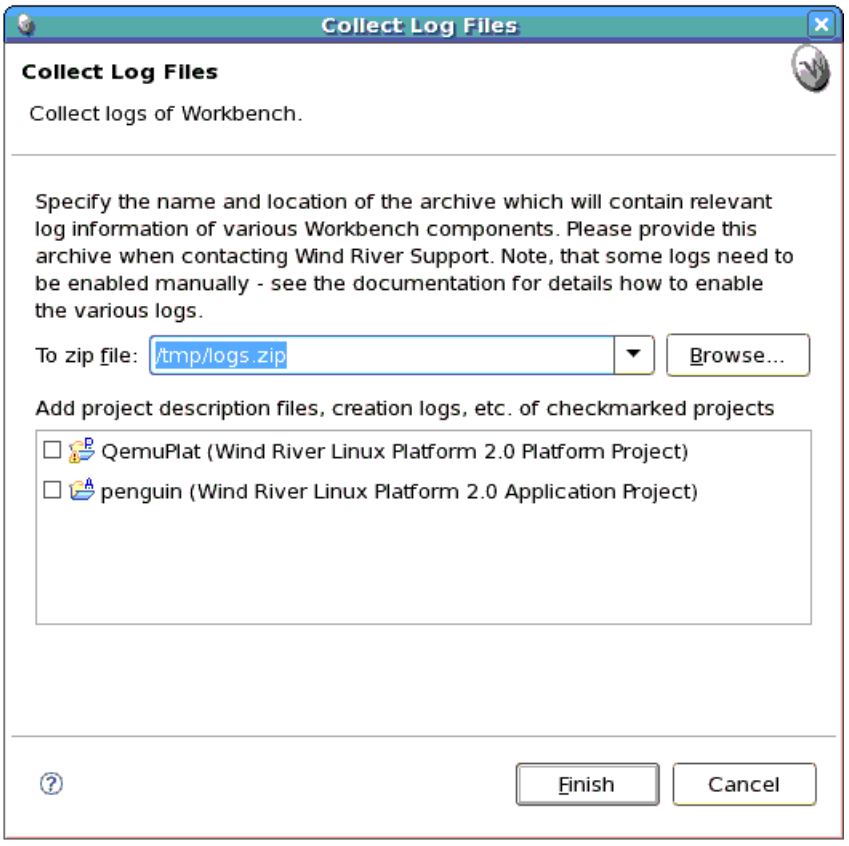
This section describes the logs, tells you how to enable them (if necessary), and how to collect them into a ZIP file you can send to Wind River support. Once all the logs you are interested in have been enabled, Workbench automatically collects the information as you work. .

NOTE: To discontinue logging for those logs that are not always enabled, clear the boxes on the Target Server Options tab, or restart Workbench without the additional options.

A.8.1 Creating a ZIP file of Logs

To create a ZIP file to send to a Wind River support representative:

1. Select **Help > Collect Log Files**. The dialog box opens.


Collect Log Files
Collect Log Files
Collect logs of Workbench.
Specify the name and location of the archive which will contain relevant log information of various Workbench components. Please provide this archive when contacting Wind River Support. Note, that some logs need to be enabled manually - see the documentation for details how to enable the various logs.
To zip file:
/tmp/logs.zip
Browse...
Add project description files, creation logs, etc. of checkmarked projects
QemuPlat (Wind River Linux Platform 2.0 Platform Project)
penguin (Wind River Linux Platform 2.0 Application Project)
Finish
Cancel

2. Type the full path and filename of the ZIP file you want to create (or browse to a location and enter a filename).
3. Check a platform project to include its build log and **config.log**.
4. Click **Finish**.

   Workbench creates the ZIP file in the specified location.

A.8.2 Eclipse Log

The information displayed in the *A.7 Error Log View*, p.355 is a subset of this log's contents.

How to Enable Log

This log is always enabled.

What is Logged

- All uncaught exceptions thrown by Eclipse java code.
- Most errors and warnings that display an error dialog box in Workbench.
- Additional warnings and informational messages.

What it Can Help Troubleshoot

- Unexpected error alerts.
- Bugs in Workbench java code.
- Bugs involving intercomponent communication.

Supported?

Yes.

A.8.3 DFW GDB/MI Log

The GDB/MI log is a record of all communication between the debugger back end (the "debugger framework," or DFW) and other views within Workbench, including the Remote Systems, debugger, and OCD views.

How to Enable Log

This log is always enabled.

What is Logged

All commands sent between Workbench and the debugger back end.

What it Can Help Troubleshoot

Debugger and Remote Systems view-related bugs.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

A.8.4 DFW Debug Tracing Log

How to Enable Log

This log is always enabled.

What is Logged

Internal exceptions in the debugger back end.

What it Can Help Troubleshoot

The debugger back end.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

A.8.5 Debugger Views GDB/MI Log

How to Enable Log

You must enable this log before you start Workbench. Do this by adding these parameters to the Workbench executable command:

```
-vmargs -DDFE.Debug=true
```

What is Logged

Same as *DFW GDB/MI Log*, p.357, except with Workbench time-stamps.

What it Can Help Troubleshoot

Debugger and Remote Systems view-related bugs.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

A.8.6 Debugger Views Internal Errors Log

How to Enable Log

You must enable this log before you start Workbench. Do this by adding these parameters to the Workbench executable command:

```
-vmargs -DDFE.Debug=true
```

What is Logged

Exceptions caught by the Debugger views messaging framework.

What it Can Help Troubleshoot

Debugger views bugs.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

A.8.7 Debugger Views Broadcast Message Debug Tracing Log

How to Enable Log

You must enable this log before you start Workbench. Do this by adding these parameters to the Workbench executable command:

```
-vmargs -DDFE.Debug=true
```

What is Logged

Debugger views internal broadcast messages.

What it Can Help Troubleshoot

Debugger views bugs.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

A.8.8 Target Server Output Log

This log contains the messages printed by the target server while running. These messages typically indicate errors during various requests sent to it, such as load operations. Upon startup, if a fatal error occurs (such as a corefile checksum mismatch) then this error will be printed before the target server exits.

How to Enable Log

- Enable this log from the Remote Systems view by right-clicking the target connection, selecting **Properties**, selecting the **Target Server Options** tab, then clicking **Edit**.

  Select the **Logging** tab, then check the box next to **Enable output logging** and provide a filename and maximum file size for the log. Click **OK**.

- Enable this log from the command line using the **-l** *path/filename* and **-lm** *maximumFileSize* options to the target server executable. For more information about target server commands, see **Wind River Documentation > References > Host API and Command References > Wind River Host Tools API Reference > tgtsvr**.

What is Logged

- Fatal errors on startup, such as library mismatches and errors during exchange with the registry.
- Standard errors, such as load failure and RPC timeout.

What it Can Help Troubleshoot

- Debugger back end
- Target Server
- Target Agent

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

A.8.9 Target Server Back End Log

This log records all requests sent to the WDB agent.

How to Enable Log

- Enable this log from the Remote Systems view by right-clicking the target connection, selecting **Properties**, selecting the **Target Server Options** tab, then clicking **Edit**.

  Select the **Logging** tab, then check the box next to **Enable backend logging** and provide a filename and maximum file size for the log. Click **OK**.

- Enable this log from the command line using the **-Bd** *path/filename* and **-Bm** *maximumFileSize* options to the target server executable. For more information about target server commands, see **Wind River Documentation > References > Host API and Command References > Wind River Host Tools API Reference > tgtsvr**.

What is Logged

Each WDB request sent to the agent. For more information about WDB services, see **Wind River Documentation > References > Host API and Command References > Wind River WDB Protocol API Reference**.

What it Can Help Troubleshoot

- Debugger back end
- Target Server
- Target Agent

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

A.8.10 Target Server WTX Log

This log records all requests sent to the target server.

How to Enable Log

- Enable this log from the Remote Systems view by right-clicking the target connection, selecting **Properties**, selecting the **Target Server Options** tab, then clicking **Edit**.

  Select the **Logging** tab, then check the box next to **Enable WTX logging** and provide a filename and maximum file size for the log. Click **OK**.

- Enable this log from the command line using the **-Wd** *path/filename* and **-Wm** *maximumFileSize* options to the target server executable. For more information about target server commands, see **Wind River Documentation > References > Host API and Command References > Wind River Host Tools API Reference > tgtsvr**.

What is Logged

Each WTX request sent to the target server. For more information about WTX services, see **Wind River Documentation > References > Host API and Command References > WTX Reference > wtxMsg.**

What it Can Help Troubleshoot

- Debugger back end
- Target Server
- Target Agent

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

A.8.11 Remote Systems View Debug Tracing Log

How to Enable Log

You must enable this log before you start Workbench. Do this by adding these parameters to the Workbench executable command:

```
-debug -vmargs -Dcom.windriver.ide.target.DEBUG=1.
```

What is Logged

Remote Systems view's internal debug errors.

What it Can Help Troubleshoot

Inconsistencies in the debugger back end.

Supported?

No. You may send this log to Wind River support, but no instructions are provided for how to interpret the information contained in it.

A.9 Technical Support

If you have questions or problems with Workbench or with Wind River Linux after completing the above troubleshooting section, or if you think you have found an error in the software, please contact the Wind River Customer Support organization (contact information is listed in the release notes for your platform). Your comments and suggestions are welcome.

B

Workbench and Command-Line Techniques

The following explores some of the ways that Workbench and the command-line are interchangeable and complementary, so that you can use the right tool for the task, or even use one to help debug the other.

Workbench and the Command Shell Environment

Setting the Environment with wrenv

The **wrenv** tool is the key to setting up a proper environment. This is the tool that gathers the environment settings from the installed products, for example when Workbench executes a build command, or when Workbench itself is launched.

NOTE: If you use the Wind River Linux make file system, an environment is automatically set up for each make command. If however you wish to try things outside of those Makefiles, you will need to set up the Wind River Linux environment using this method.

The standard usage for the **wrenv** tool is:

```
$ ./wrenv.sh -p package-name
```

where *package-name* is the package name as it appears in the **install.properties** file. For Wind River Linux 2.0, the command is:

```
$ ./wrenv.sh -p wrlinux-2.0
```

This spawns a sub-shell with the environment for the selected *package-name*, in this case **wrlinux-2.0**. In this shell the Wind River Linux tool chain and other required environment variables will be preset.

See the documentation for **wrenv** to learn about other options and usages.

Setting the Environment with Workbench

To open a shell for a specific project and have its environment automatically preset:

1. Select a project in the Project Explorer.
2. Select **Project > Open Shell**.

You now have a shell in that project's directory with the Wind River Linux environment. You can stay with this simple shell, or open a preferred shell like **sh** on Windows, or **bash** and **csh** on Solaris and Linux.

If this is a platform project, you can **cd** over to its configured project directory as follows:

```
$ cd ../project_name_prj
```

Executing Workbench Build Commands

Workbench and its build system are designed to allow the build process to be as simple and automatic as possible.

Sometimes it is valuable to take a build command from Workbench and run it on the command-line. Reasons may include:

- There may be a very complicated build issue, perhaps with a new BSP or a makefile, where you want to explore and modify the command and its environment.
- You may wish to instrument the makefiles to print or capture trace and debug information on the fly.
- You may want to experiment with build flags without endangering the existing build specs.

Here is an example of how to do this:

1. In Workbench, execute the build you wish to examine. For example, here is the build command for the **mthread** sample project as it appears in the Build Log:

```
echo  "building fsl_8548cds-glibc_cgl_DEBUG/mthread.o
";powerpc-wrs-linux-gnu-e500v2-glibc_cgl-gcc -g -fno-omit-frame-pointer
-mpowerpc -mcpu=8548 -pipe     -MD -MP -c -DARCH=ppc -o
"fsl_8548cds-glibc_cgl_DEBUG/mthread.o" "mthread.c"
```

2. Copy the command from the build log.

NOTE: When you cut a long command from Workbench, line breaks are inserted. You may need to paste this text into a text editor and remove those line breaks, and then re-copy the line from the text editor.

3. Open a shell, as described in *Setting the Environment with Workbench*, p.366. This gives you the compatible shell environment. You can run the **env** command to see all the provided environment settings that your **make** commands will inherit.

   If you are on a Windows host, you will need to start a **sh** shell so that you have a proper make environment.

   ```
   c:\WindRiver\workspace\mthread> sh
   sh-2.04$
   ```

4. Paste the command in the shell, and press **Enter**.

   ```
   sh-2.04$ echo "building fsl_8548cds-glibc_cgl_DEBUG/mthread.o
   ";powerpc-wrs-linux-gnu-e500v2-glibc_cgl-gcc -g -fno-omit-frame-pointer
   -mpowerpc -mcpu=8548 -pipe     -MD -MP -c -DARCH=ppc -o
   "fsl_8548cds-glibc_cgl_DEBUG/mthread.o" "mthread.c"
   ```

   You should get the exact same build results as you would get executing this in Workbench.

5. You can now experiment with the command and its environment to troubleshoot problems.

Executing Shell Commands in Workbench

Just as you can test Workbench commands in a shell, you can add shell or script commands to Workbench. Reasons to do this include:

- You want to have housekeeping scripts available in Workbench.
- You want to have scripts that execute in the correct and specific environment.
- You want to add temporary debug scripts that are project specific.

The following example assumes you want a command that displays all environment settings that contain the project's name.

1. Right-click a platfrom project and select **Properties > Build Properties > Build targets**.
2. In the **Custom build targets** section, click **New**.
3. Enter the name **Display Environment**.

4. Click **Variables**, select **Project Name**, and click **OK**.
5. Finish the command so that it looks like this:

```
sh -c "env | grep %selected_project_name%"
```

6. Set the **Type** to **Command**.
7. Set **Refresh after execution** to **Nothing**.
8. Click **OK** for the Build Target dialog box.
9. Click **OK** for the Properties dialog box.

Now try the new command:

1. Right-click the project and select **Build Options > Display Environment**.
2. Observe the output in the Build Log. Note that only the **PWD** setting from the environment has the project's name in it.

```
...
Command sh -c env | grep samplePlatformProject
...
PWD=/home/user/WindRiver/Workspace/samplePlatformProject
```

This is about the most complicated shell command you should enter in this dialog. It is generally better to have the command execute a script and pass parameters to it, using hand selected parameters, ones added from the project's variables as above, or ones from the project's macro list.

Setting the Environment Without Starting a Sub-shell

In some cases it is not desirable to start a sub-shell, for example:

- You are executing a script, and do not want it to have to start a sub-shell.
- You have **.cshrc** or **.profile** files that override **PATH** or other variables whenever a sub-shell is started.

NOTE: Most command-line Linux developers prefer to have absolute control over their environment settings. This usage allows such developers to choose exactly when and what to add to the given shell for the task at hand.

Exporting Environment Settings to the Current Shell

The following command-line can be used to output the environment variables to standard out:

```
$ ./wrenv.sh -p wrlinux-2.0 -p print_env -f shell-type
```

The possible values for *shell-type* include **plain**, **sh**, **csh**, **bat**, and **tcl**. A more concrete example setting environment in **csh** without starting a sub-shell:

```
$ eval `wrenv.sh -p wrlinux-2.0 -p print_env -f csh`
```

Extracting environment settings

The following command-line can be used to output the environment variables to standard out

```
$ ./wrenv.sh -p wrlinux-2.0 -p print_env | path-label
```

For example, suppose you had a script that needed to find all packages that added a path for **WIND_WRLINUX_LAYERS**. These paths would be in i**nstall.properties** in the form:

```
ipnet60.eval.01=addpath WIND_WRLINUX_LAYERS $(WIND_HOME)$/ipnet-6.0
```

To extract these paths, you could run this command:

```
$ ./wrenv.sh -p wrlinux-2.0 -p print_env | WIND_WRLINUX_LAYERS
```

NOTE: In Windows, separate the paths by the colon (**:**) character. For Unix and Linux, separate them by the semi-colon (**;**) character. If there are no matches, an empty string is returned.

C

Installing on Network Servers

Workbench and Wind River Linux can be installed on a network server instead of locally. This allows you to reduce the number of installations, simplify maintenance, and save disk space. You can also install Wind River Linux on a network server and then use a local installation of Workbench to access it.

You should consider the following issues when making an installation of Wind River Linux for use on your network:

- **You must use the same absolute path when installing and using the shared installation.**

  For example, you could mount an NFS-export from the network server locally on a Linux host on **/net**. You then **cd** to the installation directory under **/net** and install Workbench and Linux there. To run Workbench, you would reference the same directory, for example:

  ```
  $ cd /net/installDir
  $ ./startWorkbench
  ```

  Do *not* log on to the remote server and run Workbench using the path that is local to that server.

- **You must create a .wind directory at the shared location that is accessible to all users.**

  This is required for the Remote Systems view.

NOTE: You can make the rest of the installation read-only if you like—only the .**wind** directory must allow general access.

Note that Workbench will continue to use the **.workbench-***version* in your home directory and that you can also have a local workspace.

- **You must install Workbench on the shared installation server, and you must install it first.**

  This applies even if you are going to be using a local installation of Workbench. The Workbench installation on the server provides some necessary support for the Wind River Linux installation that you perform after Workbench is installed.

- **You must use the appropriate installation license key.**

  Do not, for example, use a key for a platform development installation if you are installing an application development environment because this will install the wrong infrastructure files.

- **You must have unique installations for each host operating system.**

  You cannot use the same shared installation for multiple host operating systems. Each host type must have its separate shared installation to get the proper host tools and infrastructure details.

Shared Wind River Linux Installation

Use the following procedure to install Workbench and Wind River Linux on your network server.

1. On the local host (Windows, Linux, or Solaris), install Workbench in the shared location using the proper license key.
2. On the local host, install Wind River Linux in the same location using the same key.
3. Use Workbench from any host of the same type (Windows, Linux, or Solaris).
4. Specify a local workspace.

Local Workbench Installation

You may want to use a local installation of Workbench to reduce network traffic and dependencies, or for performance reasons. Use the following procedure to install a local copy of Workbench that uses the shared Wind River Linux installation.

1. Install Workbench on your local host.
2. Append the contents of the *installDir*/**install.properties** file, located in the **wrlinux-***version* directory of the shared installation, to your local *installDir*/**install.properties** file.
3. Edit the **WIND_LX_HOME** value in your local *installDir*/**install.properties** file to point to the shared *installDir*. Note that you must always use **$/** as the directory delimiter, even for Windows hosts (see the **wrenv** documentation for details). For example:

   Before:

   ```
   wrlinux14.eval.02=export WIND_LX_HOME=$(builtin:InstallHome)
   ```

   After:

```
wrlinux14.eval.02=export WIND_LX_HOME=$/net$/sharedLinuxPlatDevDir$/WindRiver
```

You can now use this local Workbench and it will automatically use the data from the shared installation.

NOTE: If you install an additional Wind River product in this local directory, you must repeat the modifications to the local **install.properties** file described in this example.

D

Configuring a Wind River Proxy Host

D.1 Introduction 375

D.2 Configuring wrproxy 377

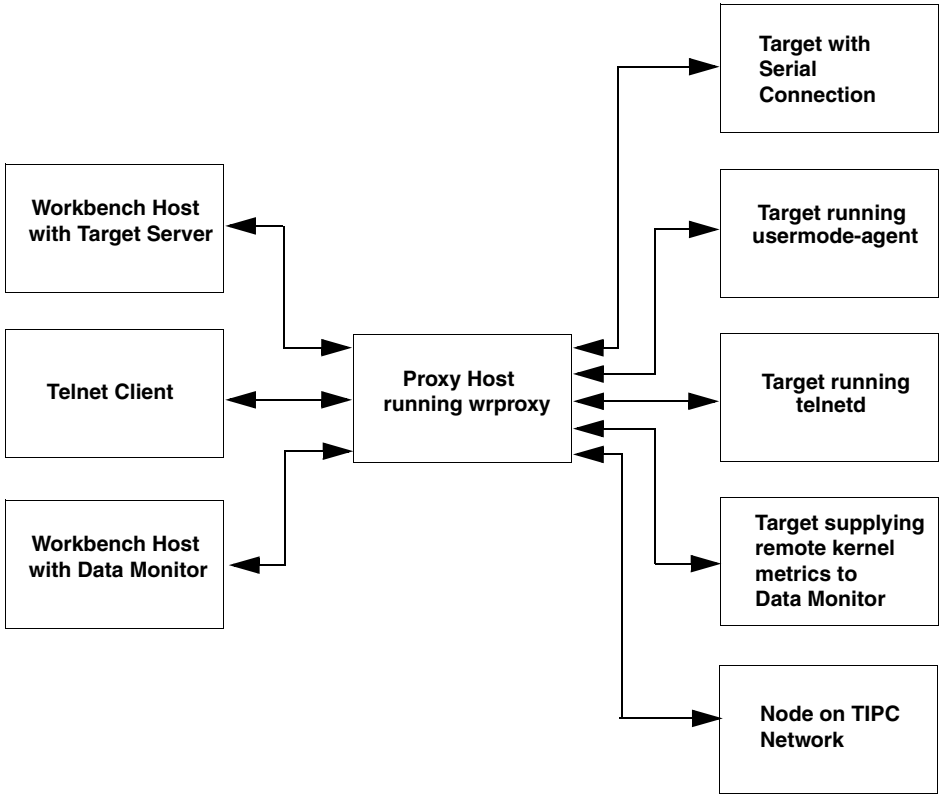
D.3 wrproxy Command Summary 379

D.1 Introduction

The Wind River proxy allows you to access targets not directly accessible to your Workbench host. For example, you might run the proxy server on a firewall and use it to access multiple targets behind the firewall.

The proxy supports TCP, UDP, and TIPC (Linux only) connections with targets. Many different host tools and target agents can be connected. A simple illustration of this is shown in Figure D-1.

Figure D-1 **Wind River Proxy Example**


Target with Serial Connection
Workbench Host with Target Server
Target running usermode-agent
Telnet Client
Proxy Host running wrproxy
Target running telnetd
Workbench Host with Data Monitor
Target supplying remote kernel metrics to Data Monitor
Node on TIPC Network

The proxy host itself can be one that runs any operating system supported for Workbench hosts or any host running Wind River Linux. You run the **wrproxy** command supplied with Workbench on the proxy host and configure it to route access from various tools to specific targets. The mapping is done by TCP/IP port number, so that access to a particular port on the proxy host is directed to a pre-defined target. You can start **wrproxy** and then manually configure it, or you can create a configuration script that **wrproxy** reads at startup.

D.2 Configuring wrproxy

The **wrproxy** command (or **wrproxy.exe** on Windows) is located in *installDir*/**workbench-***version*/**foundation/***version*/**x86-***version*/**bin/**. Copy it to the host that will serve as your proxy host. The following discussion assumes you have copied **wrproxy** to your proxy host and are configuring it from the proxy host.

Configuring wrproxy Manually

To configure **wrproxy** manually, start it with a TCP/IP port number that you will use as the proxy control port, for example:

```
$ ./wrproxy -p 1234 &
```

You can now configure **wrproxy** by connecting to it at the specified port.

Use the **create** command to configure **wrproxy** to map client (host tool) accesses on a proxy port to a particular target. The following example configures accesses to the proxy port 1235 to connect to the Telnet port of the host **my_target**:

```
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
create type=tcpsock;port=23;tgt=my_target;pport=1235
ok pport=1235
```

(Refer to *create*, p.381 for details on **create** command arguments.)

If you now connect to the proxy host at port 1235, you are connected to the Telnet port of **my_target**:

```
$ telnet localhost 1235
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

my_target login:
```

Creating a wrproxy Configuration Script

If you are typically using the same Wind River proxy configurations over time, it can be useful to use a startup script to configure it rather than doing it manually each time. You can cause **wrproxy** to read a startup script by invoking it as **wrproxy -s** *startupscript*. The script contains the commands that configure **wrproxy** as well as comments that begin with the **#** character. A simple startup script that configures the same port setup performed manually in the previous example might look like this:

```
# This is an example of a wrproxy startup script

# Configure the proxy host port 1235 to connect to my_target Telnet

create type=tcpsock;port=23;tgt=my_target;pport=1235

# list the port configuration

list

# end of script
```

When you start **wrproxy** with this script, it gets configured as in the previous example and sends input and output to standard output:

```
$ ./wrproxy -s wrproxy_startup &
[2] 6660
Executing startup script...

create type=tcpsock;port=23;tgt=my_target;pport=1235
ok pport=1235
list
ok pport=1235;type=tcpsock;port=23;tgt=my_target
$
```

Since no control port was specified with the **-p** option at startup, the default port 17476 is used.

NOTE: There is no password management in **wrproxy**. If you want to be sure that no new connections (tunnels) are made remotely using the control port, use the **-nocontrol** option with the **-s** *startupscript* option which will disable the proxy control port.

The startup script accepts the **create**, **list**, and **delete** commands as described in *Configuration Commands*, p.379.

D.3 wrproxy Command Summary

The following section summarizes all of the Wind River proxy commands.

NOTE: For all commands, unknown parameters are ignored: they are not considered errors. In addition, the client should not make any assumption on the number of values returned by the command as this could be changed in the future. For example, the **create** command always returns the value for **pport** but more information could be returned in a future version of the Wind River proxy.

Invocation Commands

The **wrproxy** command accepts the following startup options:

- **-p**[**ort**] – specify TCP control port. If not specified, the default of **0x4444** (17476) is used. This should be a unique number less than 65536 not used as a port by any other application, and it should be greater than 1024 which is the last of the reserved port numbers.
- **-V** – enable verbose mode.
- **-v**[**ersion**] – print **wrproxy** command version number.
- **-s** *startupscript* – specify a startup script that contains **wrproxy** configuration commands.
- **-h**[*elp*] – print **wrproxy** command help.
- **-nocontrol** – disable control port.

Configuration Commands

You can use the following commands interactively, and all except the **connect** command in a Wind River proxy startup script.

connect

Create a new Wind River proxy connection and automatically connect to it. Unlike the create command (see *create*, p.381) the connection is established immediately and all packets sent to the connection are immediately routed between the target and host.

Usage

connect type=*type***;mode=***mode***;proto=***proto* ; *connection_specific_parameters*

Where the arguments to the connect command are as follows:

type is:

- **udpsock** – UDP socket connection
- **tcpsock** – TCP socket connection
- **tipcsock** – TIPC socket connection (Linux only)

mode describes how the connection is handled between the proxy and the client (for example the Workbench host) and is:

- **raw** – raw mode (default)
- **packet** – packet size is sent first followed by packet content; the packet is handled only when fully received

proto describes how the connection is handled between the proxy and the target and is:

- **raw** – proxy does not handle any protocol (default).
- **wdbserial** – (VxWorks targets only) proxy converts packet to wdbserial. When **proto** is **wdbserial**, some control characters are inserted by the proxy in the packet sent to the target so that the generated packet will be understood correctly by the target using a WDB serial backend. This is typically used to connect to a WDB agent running on a target through a serial line that is connected to the serial port of a port server (this serial line is then accessible by the proxy using a well-known TCP port of the port server).

Connection-specific Parameters

- **udpsock** and **tcpsock** connection:

  port=*port***;tgt=***tgtAddr*

  Where *port* is the TCP/UDP port number and *tgtAddr* is the target IP address.

- **tipcsock** connection (Linux only):

  tipcpt=*tipcPortType***;tipcpi=***tipcPortInstance***;tgt=***tgtAddr*

  Where *tipcPortType* is the TIPC port type, *tipcPortInstance* is the TIPC port instance and *tgtAddr* is the TIPC target address.

The response of the Wind River proxy to the **connect** command is a string as follows:

```
ok
```

or

```
error errorString
```

where *errorString* describes the cause of the error.

create

Create a new proxy port mapping to a target. The connection is not established immediately as with the connect command (see *connect*, p.379) but only when a client connects to the specified port number.

Usage

`create type=`*type*`;port=`***port*`;tgt=`***target*`;pport=`***pport***

where the arguments to the **create** command are as follows:

type=*type* is:

- **udpsock** – UDP socket connection
- **tcpsock** – TCP socket connection (only **tcpsock** is allowed for a VxWorks proxy host.)
- **tipcsock** – TIPC socket connection

port – the port to connect to on the target

NOTE: If you do not assign a port number, the default value of **0x4444** is used.

tgt=*target* – host name or IP address of the target when **type** is **tcpsock** or **udpsock**, and **port** provides the UDP or TCP port number. When **type** is **tipcsock** this is the target TIPC address, and **tipcpi** provides the TIPC port instance and **tipcpt** provides the TIPC port type.

pport=*proxy_TCP_port_number* – specify the TCP port number that clients (host tools) should connect to for connection to *target_host*. This should be a unique number less than 65536 not used as a port by any other application, and it should be greater than 1024 which is the last of the reserved port numbers.

NOTE: If you do not specify a **pport** value, one will be assigned automatically and returned in the command output.

port=*target_TCP_port_number* – specify the TCP port to connect to on the target. This should be a unique number less than 65536 not used as a port by any other application, and it should be greater than 1024 which is the last of the reserved port numbers.

A simple example of using the **create** command to configure a Telnet server port connection is given in *D.2 Configuring wrproxy*, p.377.

delete

Delete the proxy configuration for a specific port.

Usage

delete pport=*port_number*

To delete the proxy configuration of a specific port, use the **delete** command with the port number, for example:

```
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
delete pport=1235
ok^]
telnet> q
Connection closed.
```

list

List your current configuration with the **list** command.

Usage

list

For example, to list your current configuration, connect to the proxy control port and enter the **list** command:

```
$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
list
ok pport=1235;type=tcpsock;port=23;tgt=my_target
```

E

Command-line Updating of Workspaces

The Workbench installation includes a **wrws_update** script that allows you to update workspaces from the command-line. This can be used, for example, to update workspaces in a nightly build script. The following section provides a reference page for the command.

A script for updating an existing workspace is available in the Workbench installation and is named:

wrws_update.bat (Windows only)

wrws_update.sh (Windows, Linux, and Solaris)

This script launches a GUI-less Eclipse application that can be used to update makefiles, symbols (source analysis), and the search index.

Execution

Specify the location of the **wrws_update** script or add it to your path and execute it with optional parameters, for example:

```
$ wrws_update.sh -data workspace_dir
```

NOTE: The workspace must be closed for the command to execute. This includes closing all instances of the Workbench GUI that are accessing this workspace.

If you do not specify any options to the command, all update operations are performed (**-all projects**, **-generate makefiles**, **--update symbols**, **-update index**).

Options

General Options

-h, --help
Print command help.

-q, --quiet
Do not produce standard output.

Eclipse Options

-data *workspace_dir*
The script uses the default workspace (if known), but it can also update other workspaces by specifying the **-data** *workspace_dir* option, just as Workbench does. (The script accepts the same command-line options as Workbench. For example, to increase virtual memory specify **-vmargs -Xmx***mem_size*.)

Global Options

-a, --all-projects

Update all projects, this option will force all closed projects to be opened. Opened projects will be closed after finishing the update.

-l, --specify-list-of-projects *argument*
Specify a list of projects to be updated. This option reduces the scope of the nightly update to the specified list of projects. Needed closed projects will be opened and unneeded opened ones closed. After finishing the update the previous state is restored. Separate the list with "," for example: **cobble,helloWorld**.

If the build target of a managed build project depends on files, folders, or sub-targets from other projects in the workspace, they must be included in the list of projects.

For example, a project named **ManagedBuildProj** references build targets from a subproject, **DependProj1**, and source files from another project, **DependProj2** (flexible managed build only). Therefore, they must be included in the list of projects, as shown here on Windows:

```
$ wrws_update -data C:\build -l DependProj1,DependProj2,ManagedBuildProj -m
```

Build Options

-b, **--build-projects** *argument*
Launch build for projects. Several strings are valid as arguments, including: **build** (default), **clean**, and **rebuild**. All open projects in the workspace are built in the correct build order. It is not required to specify a list of projects using the **-l** option.

-e, **--enableTraceBuild**
Enable trace build output.

-f, **--debugMode** *argument*
Build using specific debug or non-debug mode where applicable. The *argument*, if specified, can be **0** or **1**, otherwise the current mode is used per project.

-u, **--buildArgs** *argument*
Specify a list of additional build options. Separate the list with "," for example: **-i,MY_VAR=***value*.

Launch Options

-r, **--run-config** *argument*
Run the launch configuration named *argument* previously created using the Workbench GUI. Execution will always be in run mode. If the launch configuration name has spaces within the name, you must enclose the name in double quotes and escape the enclosing quotes (for example, **-r \"launch config name with spaces\"**).

Nightly Update Options

-i, **--update-index**
Update search-database index.

-m, **--generate-makefiles**
Regenerate makefiles where necessary.

-s, **--update-symbols** *argument*
Update symbol database (source analysis). To create the data from scratch, you can supply **rebuild** as *argument*.

-t, **--create-team-symbols** *argument*
Export symbol databases for shared use in a team. The argument is a quoted comma-separated list of options. Valid options are **timestamp**, **readonly**, and

checksum. The default is **timestamp,readonly,checksum**. See the online documentation for details on these options.

-x, **--update-xref** *argument*
Update cross references (source analysis). To create the data from scratch, you can supply **rebuild** as *argument*.

Output

Any errors that might occur during the updates are printed out to standard error output. Other information (for example, status, what has been done, and so on) are printed out to standard output.

NOTE: No configuration management-specific actions or commands are executed within this script and the launched application. Configuration management specific synchronizations or updates relevant to the workspace (for example, **cvs-update**, ClearCase view synchronization, and so on) have to be done before this script is started.

F

Command-line Importing of Projects

F.1 Introduction 387

F.2 wrws_import Reference 388

F.1 Introduction

The Workbench installation includes a **wrws_import** script that allows you to import existing projects into workspaces from the command-line. The following section provides a reference page for the command.

F.2 wrws_import Reference

A script for launching a GUI-less Eclipse application that can be used to import existing projects into the workspace is available in the Workbench installation and is named:

wrws_import.bat (Windows only)

wrws_import.sh (Windows, Linux, and Solaris)

Execution

Specify the location of the **wrws_import** script or add it to your path and execute it with optional parameters, for example:

```
$ wrws_import.sh -data workspace_dir
```

Options

General Options

-d, --debug *argument*
Provide more information. The argument, if given, specifies the level of verbosity. Default is **2**, the possible options are: [**2, 3, 4**].

-h, --help
Print command help.

-q, **--quiet**
Do not produce standard output.

Eclipse Options

-data *workspace_dir*
Specify the Eclipse workspace with this option.

Import Project Options

-f, **--files** *argument*
Specify a list of project files to be imported. Separate the items in the list with commas (**,**). For example: *dir1***/.project**,*dir2***/.project**. All files must be specified using an absolute path.

-r, **--recurse-directory** *argument*
Specify a directory to recursively search for projects to be imported. All files must be specified using an absolute path.

-v, **--define-variables** *argument*
Specify a list of Eclipse path variables to be defined. Separate the entries in the list with commas (**,**). For example: *pathvar1*=*path1*,*pathvar2*=*path2*.

NOTE: This script will not stop or fail if some projects already exist in the Workspace, the way the **Import existing projects into workspace** wizard does. It will just print out the information and continue.

G

Starting usermode-agent at Boot Time

This appendix contains an example script for starting usermode-agent when the Linux target system boots.

To do user mode debugging, you must run the user mode agent on the Wind River target. The agent becomes installed in the root file system by default, but without this script or an equivalent, you have to start it manually. The following script:

- Installs the user mode agent as an OS service and
- Enables automatic startup via the **init** mechanism when booting

You may place this script in **/etc/rc3.d** in the target file system.

```
#!/bin/sh
#
# usermode-agent          This script takes care of starting and stopping UMA
#
#

prog="usermode-agent"

start() {
        # Attach usermode-agent device
        echo -n $"Starting $prog: "
        echo `/usr/bin/usermode-agent -v`
        /bin/sh -c "/usr/bin/usermode-agent -v" > /tmp/usermode-agent.version
2>&1 &
        /bin/sh -c "/usr/bin/usermode-agent -V" > /tmp/usermode-agent.log
2>&1 &
    echo "[  OK  ]"
    touch /var/lock/subsys/usermode-agent
        echo
}
```

```
stop() {
        # Stop service.
        echo -n $"Shutting down $prog: "
    killproc usermode-agent
    rm -f /var/lock/subsys/usermode-agent
        echo
}

# See how we were called.

case "$1" in
  start)
    start
        ;;
  stop)
    stop
        ;;
  restart|reload)
    stop
    start
    ;;
  condrestart)
    [ -e /var/lock/subsys/usermode-agent ] && (stop; start)
    ;;
  *)
        echo $"Usage: $0 {start|stop|restart|reload|condrestart}"
        exit 1
esac

exit 0
```

H

Glossary

This glossary contains terms used in Wind River Workbench. For basic Eclipse terms, see the Eclipse glossary.[1]

If the term you want is not listed in one of these glossaries, you can search for it throughout all online documentation.

1. At the top of the **Help > Help Contents** window, type your term into the **Search** field.
2. Click **Go**. Topics containing the term will appear in the **Search Results** list.
3. To open a topic in the list, click it.

For more information about online help, see **Help > Help Contents > Wind River Documentation > Eclipse Platform Documentation > Eclipse Workbench User Guide > Tasks > Using the help system**.

active view

The view that is currently selected, as shown by its highlighted title bar. Many menus change based on which is the active view, and the active view is the focus of keyboard and mouse input.

1. To access the Eclipse glossary, see **Help > Help Contents > Wind River Documentation > Eclipse Platform Documentation > Eclipse Platform Plug-inDeveloper Guide > Reference > Other reference information > Glossary of Terms**.

back end

Functionality configured into a target server which allows it to communicate with various target agents, based on the mode of communication that you establish between the host and the target (network, serial, and so on).

build

The type of project built: *managed build* by default (formerly *flexible build*); and also a deprecated but still available *standard managed build* (sometimes known as a *standard build*). There are also *user-defined* and *disabled builds*.

build output analysis

Also referred to as discovery, a mechanism to analyze build output to determine include paths, symbols, and the build scope (affected files) during a build run.

build scope

The set of files built during a build run, determining which source files are processed by the indexer. Generally, all projects use a build-driven setup, unless build information is missing, in which case, all files will be parsed (indexed).

CDT (C/C++ Development Tooling

The Eclipse C/C++ IDE.

code completion

A feature of the source editor, by which Ctrl+Space presents a pop-up list of proposals suitable for the current cursor position. Proposals include types, methods, functions of the platform API (depending on the project type, and referred to as external API), and types, methods, functions, and variables that are visible in the current scope.

color context

The color assigned to a particular process in the Debug view; this color carries over to breakpoints in the Editor and to other views that derive their context from the Debug view.

cross-development

The process of writing code on one system, known as the host, that will run on another system, known as the target.

debuggable objects

Debuggable objects are the executable application files, kernels, kernel modules, and libraries that can be accessed by both the host and Linux target. These objects are ideally compiled without optimization, compiled with the appropriate debug flags (for example with **-g**, or **-g-dwarf-2**), and are not stripped of symbols.

disabled build

Project builds for which Workbench provides no build support at all. Useful for projects or folders that contain, for example, only header or documentation files that do not need to be built.

discovery

See *build output analysis*.

editor

An editor is a visual component within Wind River Workbench. It is typically used to edit or browse a file or other resource.

Modifications made in an Editor follow an open-save-close life cycle model. Multiple instances of an Editor type may exist within a Workbench window.

element

Aprogramming language element, such as a class, structure, union, typedef, enumeration, namespace, function, method, or variable (includes member variables of types). *Element* replaces the former notion of the term *symbol*.

external API

An index holding a pre-parsed platform API for a project type. Most managed project types come with an external API package which, when indexed, provides the most common API functions, methods, and types for code completion and parameter hints in the source editor.

flexible managed build

Deprecated name for what is currently called *managed build*.

help key

Press the help key in Workbench to get context-sensitive help. The help key is host-dependent. On a Windows host, press **F1**. On a Linux or Solaris host, press **CTRL+F1**.

host shell

A Wind River command shell that provides a command-line environment for GDB and KGDB debugging. The host shell also provides Tcl scripting support.

hover info

Information presented by the source editor as you move the cursor over any method, function, variable, or type. The source editor presents the signature (parameter hints) and comment of any corresponding declaration in a tool tip.

hunk

A hunk is a contiguous group of source lines generated when the **diff** program is applied to compare files. The **patch** program and the Quilt patch program based upon it use **diff** to create patches, which are then internally represented as one or more hunks to apply to a file to patch it.

index

Source analysis information gathered by the indexer by parsing source code. The index is the basis for any source navigation and editing capabilities, such as navigation betweendeclaration and definition of methods, showing included files in the Include Browser, showing the call tree of functions and methods, showing the type hierarchy of classes, code completion in the editor, hover information, etc.

JDT

Java Development Toolkit provided by the Eclipse organization (**http://www.eclipse.org**) and included with Workbench.

JNI

Java Native Interface is a means of calling non-Java code (native code) from within a Java application.

kernel mode

For Linux 2.6 and higher kernels, two connection modes are supported: kernel and user mode connections. Kernel mode connections allow **kgdb** debugging of the kernel in a manner analogous to debugging applications in user mode.

kernel module

A piece of code, such as a device driver, that can be loaded and unloaded without the need to rebuild and reboot the kernel.

launch configuration

A run-mode launch configuration is a set of instructions that instructs the IDE to connect to your target and launch a process or application. A debug-mode launch configuration completes these actions and then attaches the debugger.

managed build

A build for which Workbench controls all phases, available for all project types except user-defined projects.

native mode development environment

A development environment requiring a usermode agent program to be running on the target, in a Linux operating system. In this environment, the debugger and application are compiled with the same toolchain, thus no emulator is required when running in self-hosted mode.

A native mode development environment can only be used for application development.

object path mappings

The object path mappings specify where the debuggable objects are to be found for both the debugger running on the host and the Linux target. In Workbench, this is set within the Remote Systems view's **Target Connection Properties**.

overview ruler

The vertical borders on each side of the Editor view. Breakpoints, bookmarks, and other indicators appear in the overview ruler.

parameter hinting

See *hover info*.

perspective

A perspective is a group of views and Editors in the Workbench window. One or more perspectives can exist in a single Workbench window. Each perspective contains one or more views and Editors. Within a window, each perspective may have a different set of views but all perspectives share the same set of Editors.

plug-in

An independent module, available from Wind River, the Eclipse Foundation, or from many Internet Web sites, that delivers new functionality to Wind River Workbench without the need to recompile or reinstall it.

program counter

The address of the current instruction when a process is suspended.

project

A collection of source code files, build settings, and binaries that are used to create a downloadable application or bootable system image.

registry

The registry associates a target server's name with the network address needed to connect to that target server, thereby allowing you to select a target server by a convenient name.

self-hosted development environment

The opposite of cross-development. The debugger and the application are running on the same machine.

source lookup path

The source lookup path specifies the location that the Workbench debugger uses to identify and open each source file as it is being debugged. This is set in the Debug view in Workbench.

standard build; standard managed build

Synonymous, deprecated project build types suitable for projects with build structures similar to the file system structure.

symbol

A preprocessor macro (define). Symbols can be defined by using the -D compiler option as well as by #define statements in source code. The term *symbol* in the former meaning of programming language element (functions, methods, types, etc.) has been replaced by the term *element*.

system mode

When in system mode, the debugger is focused on kernel processes and threads. When a process is suspended, all processes stop. Compare with user mode.

target agent

The target agent runs on the target, and is the interface between Wind River Linux and all other Wind River Workbench tools running on the host or target.

target server

The target server runs on the host, and connects the Wind River Workbench tools to the target agent. There is one server for each target; all host tools access the target through this server.

TIPC

Transparent inter-process communication protocol typically used by nodes within a cluster. Wind River provides a proxy and usermode agent program that allow Workbench to access targets within the TIPC cluster.

user-defined build

Project builds for which you set up and maintain your own build system and makefiles, and for which Workbench provides minimal support beyond a GUI launch, make rules expressed in the Project Explorer, and build output to the Build Console.

user mode

When in user mode, the debugger is focused on user applications and processes. When a process is suspended, other processes continue to run. Compare with system mode. For Linux 2.6 and higher kernels, user mode is a separate connection type. Compare with kernel mode.

view

A view is a visual component within Workbench. It is typically used to navigate a hierarchy of information (like the resources in your Workbench), open an Editor, or display properties for the active Editor.

Modifications made in a view are saved immediately. Only one instance of a particular view type may exist within a Workbench window.

workspace

The directory where your projects are created. To share the build objects of your projects with a target, the workspace (directory) may be in a file system that is exported to the target, or you may redirect build objects from your workspace to a location exported to the target.

Index

A

adding
- application code to projects 162
- new files to projects 163
- subprojects 149

agent-proxy 89
Analysis Tools 120
annotations in patches 135
application project (Wind River Linux) 13
attaching
- to core file 311
- to running process 283

B

back end, target server 231
ball sample program 37
basename mappings 235
Bellard, Fabrice 61
Bookmarks tab 44
breakpoints
- conditional 259
- converting to hardware 261
- data 259
- disabling 263
- expression 259
- hardware 259
- line 259
- refreshing 262
- removing 263
- restricted 259
- unrestricted 259

Breakpoints view 257
bridging hosts 90
BSPs 81
build
- applications for different boards 204
- architecture-specific functions 207
- library for test and release 204
- make rule in Project Explorer 210
- output
  - disabling prompt to add to ClearCase 331
- remote 215
- remote connection 215
- remote, setting up environment 214
- support
  - disabled 190
- target
  - excluding with regular expressions 193
- user-defined 190

build nodes (Wind River Linux) 59
build specs
- setting 37

build_all 59

C

ClearCase
  disabling prompt to add build output files 331
  installing plug-ins 317
  with Workbench 329
colored views 293
command line
  import projects (wrws_import) 387
  update workspaces (wrws_update) 383
compiler
  flags, add 202
conditional breakpoints 259
-configuration startup option 339
connection type 14
connections
  Ethernet 95
  hardware targets 63, 64
  QEMU 61
  serial 98
  TCP 89
  telnet 97
  testing 67
  UDP 89
Console view 281
context pointer 292
cooperative debugging (Java and JNI) 302
core files
  acquiring 310
  attaching Workbench 311
  general 309
creating custom kernel module 118
cross-development concepts 11
Customer Specific
  Linux Application project 147
  Linux Kernel project 148

D

data breakpoints 259
Data Monitor 307
-data startup option 336
debug modes 294
Debug perspective 36
debug server
  loading symbols 234
Debug view 288
debugger
  CDT/GDB 35
  disconnecting and terminating processes 296
  single-stepping through code 293
deleting
  project nodes 167
deploy, automated target 138
derived resource, not adding to ClearCase 331
Device Debug perspective 34
disabled build support 190
disabling breakpoints 263
Disassembly view 301
  opening automatically 301
  opening manually 301
dist.tar.bz2 files 129

E

Eclipse
  using Workbench in 323
Editor 173
  context pointer 292
environment commands (Launch Control) 279
error condition command (Launch Control) 277
Error Log view 355, 357
Ethernet
  connections 95
  ports 93
exclusion filters 342
Exec Path on Target
  troubleshooting 352
export directories 57
export/dist directories 51
  QEMU 61
export-layer 59
export-sysroot 59
expression breakpoints 259
external locations 84

F

File Navigator view 172
file system configuration (Wind River Linux) 121
File System Layout Configuration Tool 129
file systems
  browsing target's 68
  copying files within 68
  NFS mounting 91
files
  manipulating 166
find and replace 175

G

GDB debugger 35

H

hardware
  breakpoints 259
Hello World tutorial 32
help system
  on Solaris 339
  on Windows 340

I

ICE-like debugging 110
importing
  build settings 163
  resources 162
Include Browser view 173
indexer
  preferences 176

J

Java applet 272
Java application 272
Java Development Tools (JDT), installing 321
Java project 272
Java-JNI cooperative debugging 302
JNI 302

K

kernel
  debugging with QEMU 110
  metrics 306
  reconfiguration (Wind River Linux) 116
  safe areas 92
  signals 306
Kernel Configuration (Wind River Linux) 116
Kernel Configuration node (Wind River Linux) 59
kernel mode debugging 14
kernel modules 117
  custom 118
  moving projects 119
KGDB
  and Workbench 88
  enabling 92
  enabling and disabling 113
  over Ethernet 110
KGDB-OE 94
kgdboe 89

L

launch (terminology) 276
launch configurations
  automatic 69
  creating 266
  modifying 70
Launch Control 276
launch sequence 276
launching
  programs, manually 275
layers 54, 81
  adding 83
line breakpoints 259

linked resources
path variables 192
linking project nodes, moving and 167
loading symbols to debug server
specifying an object file 234
logical nodes 166

M

make rule in Project Explorer 210
makefile
build properties 197
memory
cache size, target server 233
menuconfig 116
mode
debug type 14
modprobe commands 93
multiple
launches, controlling 276
processes, monitoring 291
target operating systems or versions 197

N

Native Application
project 13, 148, 155
application code 158
creating 156
navigation 164
New Connection wizard 228
NFS
mounting file systems 91
nodes
moving and (un-)linking project 167
resources and logical 166

O

object path mappings 107
QEMU 63
why required 234
operating systems, multiple 197

P

pango error 339
patch annotations 135
patches
managing with Quilt 133
viewing annotations 135
path variables 192
pathname prefix mappings 235
penguin 70
pkglist files 123
platform 117
kernel modules 117
platform project 50
plug-ins
activating 319
adding an extension location 319
creating a directory structure 316
creating a Workbench plug-in for Eclipse 323
installing ClearCase 317
web sites 316
post-launch command (Launch Control) 277
pre-launch command (Launch Control) 277
processes
attaching to running 283
disconnecting debugger 296
project
application code 146
build
remote 215
closing 164
create
for read-only sources 347
creating 145, 162
Customer Specific
Linux Application 147
Linux Kernel 148
external location 84
files (Wind River Linux) 60
files, version control of 330
go into 164

names 144
Native Application 13, 148, 155
opening 163
platform 50
project structures 148
scoping 164
user-defined 12, 148, 190
Wind River Linux Application 13, 147
Wind River Linux Platform 13, 147
Project Explorer
move, copy, delete 165
moving and (un-)linking project nodes 167
target nodes, manipulating 168
user-defined build-targets 210
proxy host 375

Q

QEMU
connections 61
export/dist directories 61
KGDB kernel debugging 110
Quilt
commands 134
importing packages 136
patching model 133
setting up 133
viewing patch annotations 135

R

read-only sources
creating projects for 347
redirection root directory
with ClearCase 330
registry 237
changing default 239
data storage 238
error, unreachable 337
launching the default 237
shutting down 239
wtxregd 238
regular expressions
to exclude contents of build target 193
remote
build 215
setting up environment 214
connection 215
rlogin 217
SSH 217
Java
application 272
debugging 274
launch and connect 272
kernel metrics (RKM) 306
Remote System Explorer 53, 67
Remote Systems view 228
basename mappings 235
New Connection wizard 228
object path mappings 234
pathname prefix mappings 235
shared connection configuration 236
resources and logical nodes 166
RKM see remote kernel metrics (RKM)
rlogin remote build connection 217
root file systems 63
RPMs 123
configuration (Wind River Linux) 121
Run-Time Analysis Tools 120

S

safe areas 92
sample programs
ball 37
penguin 70
searching for text 175
serial cable connections 98
set, working 165
setting breakpoints
restricted 259
unrestricted 259
signals, POSIX 80
Source Analysis
description 169
source analysis

preferences 176
source lookup path
adding sources 270
editing 297
SRPM packages 136
SSH remote build connection 217
startup, Workbench 6
subprojects
adding 149
Symbol Browser view 171
sysroots 37, 59
system mode
compared with task mode 294

T

tar.bz2 files 57
target
deployment, automated 138
file systems
contents 129
operating systems, multiple versions 197
PCD 94
servers
back end settings 231
memory cache size 233
timeout options 233
TIPC 246
USB 242
TCP connections 89
team
defining path variables 192
telnet connections 97
templates 54, 81
adding 82
text
replacing 175
search 175
TFTP, configuring 64
tgtsvr command (TIPC) 232, 250
TIPC
kernel module 247
overview 245
proxy 248
targets 246
tools, development
communications, managing 227
troubleshooting
download failed, wrong build spec 351
Error Log view 355, 357
exception on attach 351
Exec Path on Target 352
help system
on Solaris 339
on Windows 340
launch configurations 354
pango error 339
registry unreachable 337
resetting Workbench defaults 340
running a process 353
target connection 350
workspace cannot be locked 338
tutorial
ball sample program 37
Hello World 32
QEMU 61
Workbench for Linux 31–85
Type Hierarchy view 172

U

UDP connections 89
ulimit command and core files 310
Update Manager 321
USB
target connection 242
user mode debugging 14
User Space Configuration 59, 121
user-defined
builds 190
projects 12, 148, 151, 152
usermode-agent
building 85
external locations 84
hardware targets 64
reference page 251
running 65

V

version control, adding Workbench project files to 330
VIO see virtual I/O (VIO)
virtual I/O (VIO) 281
Virtual-WRLinux0 terminal window 63

W

Wind River Linux
  applications 13, 147
  platforms 13, 50, 147
Wind River proxy 375
Workbench
  Application Development perspective
    creating a project 37
  bookmarks
    creating 44
    finding 44
    removing 47
    viewing 44
  breakpoints
    modifying 80
    running to 78
    setting 78
  building a project
    build errors 46
    rebuilding 47
  ClearCase views 329
  comparing files 47
  creating a project 37
  Eclipse environment 323
  Editor
    bookmarks, removing 47
    parameter hints 42
  editors 10
  Embedded Debug perspective 73
  GUI elements 7
  help system
    on Solaris 339
    on Windows 340
  moving and sizing views 9
  Outline view
    bookmarks, creating 44
  perspectives 10
    Embedded Debug perspective 73
  project source
    bookmarks
      creating 44
      finding 44
      removing 47
      viewing 44
    breakpoints
      modifying 80
      running to 78
      setting 78
    file history, viewing 47
    Outline view 41
    parameter hints 42
    stepping into 76
    string, finding 42
    symbol, finding 41
    version control 330
    viewing 40
  rebuilding a project 47
  running sample program
    with Embedded Debug perspective 74
  starting
    workspace, specifying 6
  stepping in project source 76
  tabbed notebooks 9
  terminology 7
  viewing project source 40
  views 9
    Breakpoints 257
    colored 293
    Debug 288
    Disassembly 301
    Editor 173
    Error Log 355, 357
    File Navigator 172
    Include Browser 173
    Outline view 41
    Symbol Browser 171
    Type Hierarchy 172
  windows 8
  workspace 6
working sets 170

using 165
workspace
directories 144
starting Workbench with a new 336
wrproxy command 375
wrws_import
reference page 388
script 387
wrws_update
reference page 383
script 383
wtxregd
using a remote registry 238

X

xconfig 116